15-424: Foundations of Cyber-Physical Systems

Lecture Notes on Choice & Control

André Platzer

Carnegie Mellon University

Lecture 3

1 Introduction

In the previous lecture, we have seen the beginning of cyber-physical systems, yet emphasized their continuous part in the form of differential equations $x' = \theta$. The sole interface between continuous physical capabilities and cyber capabilities was by way of their evolution domain. The evolution domain H in a continuous program $x' = \theta \,\&\, H$ imposes restrictions on how far or how long the system can evolve along that differential equation. Suppose a continuous evolution has succeeded and the system stops following its differential equation, e.g., because the state would otherwise leave the evolution domain. What happens now? How does the cyber take control? How do we describe what the cyber elements compute and how they interact with physics?

This lecture extends the model of continuous programs for continuous dynamics to the model of hybrid programs for hybrid dynamics.

This lecture is based on material on cyber-physical systems and hybrid programs [Pla12b, Pla10, Pla08, Pla07].

Continuous programs $x' = \theta \,\&\, H$ are very powerful for modeling continuous processes. They cannot—on their own—model discrete changes of variables, however.[1] During the evolution along a differential equation, all variables change continuously in time, because the solution of a differential equation is (sufficiently) smooth. Discontinuous change of variables, instead, needs a way for a discrete change of state. What could be a model for describing discrete changes in a system?

[1]There is a much deeper sense [Pla12a] in which continuous dynamics and discrete dynamics are quite surprisingly close together. That understanding requires a lot more logic than we have at our disposal at this stage of the course. It also leads to a full understanding of what constitutes the hybridness of hybrid systems. Yet, its understanding does rest on the foundations of hybrid systems, which we need to understand first.

There are many models for describing discrete change. You will have seen a number of them already. CPSs combine cyber and physics. In CPS, we do not program computers, but program CPSs instead. As part of that, we program the computers that control the physics. And programming computers amounts to using a programming language. Of course, for programming an actual CPS, our programming language will ultimately have to involve physics. But we have already seen continuous programs in the previous lecture for that very purpose. What's missing is a way to program the discrete and cyber aspects.

Does it matter which discrete programming language we choose as a basis? It could be argued that the discrete programming language does not matter as much as the hybrid aspects do. After all, there are many programming languages that are Turing-equivalent, i.e. that compute the same functions. Yet even among them there are numerous differences for various purposes in the discrete case, which are studied in the area of Programming Languages.

For the particular purposes of CPS, however, we will find further desiderata, i.e. things that we expect from a programming language to be adequate for CPS. We will develop what we need as we go.

More information about choice and control can be found in [Pla10, Chapter 2.2,2.3].

2 Discrete Programs and Sequential Composition

Discrete change happens in computer programs when they assign a new value to a variable. The statement $x := \theta$ assigns the value of term θ to variable x. It leads to a discrete, discontinuous change, because the value of x does not vary smoothly but radically when assigning θ to x.

This gives us a discrete model of change, $x := \theta$, in addition to the continuous model of change, $x' = \theta \,\&\, H$ from the previous lecture. Now, we can model systems that are either discrete or continuous. Yet, how can we model proper CPS that combine cyber and physics and that, thus, simultaneously combine discrete with continuous dynamics?

One way how cyber and physics can interact is if a computer provides input to physics. Physics may mention a variable like a for acceleration and a computer program sets its value depending on whether the computer program wants to accelerate or brake. That is, cyber could set the values of actuators that affect physics.

In this case, cyber and physics interact in such a way that cyber first does something and physics then follows. That corresponds to a sequential composition $(\alpha; \beta)$ in which first the HP α on the left of ; runs and, when it's done, the HP β on the right of ; runs. For example, the following HP

$$a := a + 1;\; x' = v, v' = a \tag{1}$$

will first let cyber perform a discrete change of setting a to $a + 1$ and then let physics follow the differential equation $x'' = a$. The overall effect is that cyber increases a and physics then lets x evolve with acceleration a (and increases velocity v with derivative

a). Thus, HP (1) models a situation where the desired acceleration is commanded once to increase and the robot then moves with that acceleration. Note that the sequential composition operator (;) has basically the same effect that it has in programming languages like Java or C0. It separates statements that are to be executed sequentially one after the other. If you look closely, however, you will find a subtle difference in that Java and C0 expect more ; than hybrid programs.

The HP in (1) executes control (it sets the acceleration for physics), but it has very little choice. Actually no choice at all. So only if the CPS is very lucky will an increase in acceleration be the right action to remain safe.

3 Decisions in Hybrid Programs

In general, a CPS may have to check conditions on the state to see which action to take. One way of doing that is the use of an if-then-else, as in classical discrete programs.

$$\begin{aligned} &\texttt{if}(v < 4)\, a := a + 1 \,\texttt{else}\, a := -b; \\ &x' = v, v' = a \end{aligned} \tag{2}$$

This HP will check the condition $v < 4$ to see if the current velocity is still less then 4. If it is, then a will be increased by 1. Otherwise, a will be set to $-b$ for some braking deceleration constant $b > 0$. Afterwards, i.e. when the if-then-else statement has run to completion, the HP will again evolve x with acceleration a along a differential equation.

The HP (2) takes only the current velocity into account to reach a decision on whether to accelerate or brake. That is usually not enough information to guarantee safety, because a robot doing that would be so fixated on achieving its desired speed that it would happily speed into any walls or other obstacles along the way. Consequently, programs that control robots also take other state information into account, for example the distance $x - o$ to an obstacle o from the robot's position x, not just its velocity v:

$$\begin{aligned} &\texttt{if}(x - o > 5)\, a := a + 1 \,\texttt{else}\, a := -b; \\ &x' = v, v' = a \end{aligned} \tag{3}$$

They could also take both distance and velocity into account for the decision:

$$\begin{aligned} &\texttt{if}(x - o > 5 \land v < 4)\, a := a + 1 \,\texttt{else}\, a := -b; \\ &x' = v, v' = a \end{aligned} \tag{4}$$

Note 1 (Iterative design)**.** *As part of the labs of this course, you will develop increasingly more intelligent controllers for robots that face increasingly challenging environments. Designing controllers for robots or other CPS is a serious challenge. You will want to start with simple controllers for simple circumstances and only move on to more advanced challenges when you have fully understood and mastered the previous controllers, what behavior they guarantee and what functionality they are still missing.*

4 Choices in Hybrid Programs

What we learn from the above discussion is a common feature of CPS models: they often include only some but not all detail about the system. And for good reasons, because full detail about everything can be overwhelming. A (somewhat) more complete model of (4) might look as follows, with some further formula S as an extra condition for checking whether to actually accelerate:

$$\begin{aligned}&\texttt{if}(x-o>5\land v<4\land S)\,a:=a+1\,\texttt{else}\,a:=-b;\\&x'=v,v'=a\end{aligned}\tag{5}$$

The extra condition S may be very complicated and often depends on many factors. It could check to smooth the ride, optimize battery efficiency, or pursue secondary goals. Consequently, (4) is not actually a faithful model for (5), because (4) insists that the acceleration would always be increased just because $x-o>5\land v<4$, unlike (5), which checks the additional condition S. Likewise, (3) certainly is no faithful model of (5). But it looks simpler.

How can we describe a model that is simpler than (5) by ignoring the details of S yet that is still faithful? What we want this model to do is characterize that the controller may either increase acceleration by 1 or brake and that acceleration certainly only happens when $x-o>5$. But the model should make less commitment than (3) about under which circumstances braking is chosen. So we want a model that allows braking under more circumstances than (3) without having to model precisely under which circumstances that is. In order to simplify the system faithfully, we want a model that allows more behavior than (3).

Note 2 (Abstraction)**.** *Successful CPS models often include relevant aspects of the system only and simplify irrelevant detail. The benefit of doing so is that the model and its analysis becomes simpler, enabling us to focus on the critical parts without being bogged down in tangentials. This is the power of abstraction, arguably the primary secret weapon of computer science. It does take considerable skill, however, to find the best level of abstraction for a system. A skill that you will continue to sharpen through your entire career as a computer scientist.*

Let us take the development of this model this step by step. The first feature that the controller of this model has is a choice. The controller can choose to increase acceleration or to brake, instead. Such a choice between two actions is denoted by the operator $\cup$:

$$\begin{aligned}&(a:=a+1\cup a:=-b);\\&x'=v,v'=a\end{aligned}\tag{6}$$

When running this hybrid program, the first thing that happens is a choice between whether to run $a:=a+1$ or whether to run $a:=-b$. That is, the choice is whether to increase a by 1 or whether to reset a to $-b$ for braking. After this choice (i.e. after the ; operator), the system follows the usual differential equation $x''=a$.

Note 3 (Nondeterministic $\cup$)**.** *The choice* $(\cup)$ *is* nondeterministic. *That is, every time a choice* $\alpha \cup \beta$ *runs, exactly one of the two choices,* α *or* β*, is chosen to run and the choice is* nondeterministic, *i.e. there is no prior way of telling which of the two choices is going to be chosen.*

The HP (6) is a *faithful abstraction* of (5), because every way how (5) can run can be mimicked by (6) so that the outcome of (6) corresponds to that of (5). Whenever (5) runs $a := a + 1$, which happens exactly if $x - o > 5 \land v < 4 \land S$ is *true*, (6) only needs to choose to run the left choice $a := a + 1$. Whenever (5) runs $a := -b$, which happens exactly if $x - o > 5 \land v < 4 \land S$ is *false*, (6) needs to choose to run the right choice $a := -b$. So all runs of (5) are possible runs of (6). Furthermore, (6) is much simpler than (5), because it contains less detail. It does not mention $v < 4$ nor the complicated extra condition S. Yet, (6) is a little too permissive, because it suddenly allows the controller to choose $a := a + 1$ even at close distance to the obstacle, i.e. even if $x - o > 5$ is *false*. That way, even if (5) was a safe controller, (6) is still an unsafe one.

5 Tests in Hybrid Programs

In order to make a faithful and not too permissive model of (5), we need to restrict the permitted choices in (6) so that the acceleration choice $a := a + 1$ can only be chosen at sufficient distance $x - o > 5$. The way to do that is to use tests on the current state of the system.

A *test*[2] $?H$ is a statement that checks the value of a first-order formula H of real arithmetic in the current state. If H holds in the current state, then the test passes, nothing happens, yet the HP continues to run normally. If, instead, H does not hold in the current state, then the test fails, and the system execution is aborted and discarded. That is, when ν is the current state, then $?H$ runs successfully without changing the state when $\nu \models H$. Otherwise, i.e. if $\nu \not\models H$, the run of $?H$ is aborted and not considered any further.

The test statement can be used to change (6) around so that it allows acceleration only at large distances while braking is still allowed always:

$$\begin{aligned}&\big((?x - o > 5; a := a + 1) \cup a := -b\big);\\ &x' = v, v' = a\end{aligned} \tag{7}$$

The first statement of (7) is a choice $(\cup)$ between $(?x - o > 5; a := a + 1)$ and $a := -b$. All choices in hybrid programs are nondeterministic so any outcome is always possible. In (7), this means that the left choice can always be chosen, just as well as the right one. The first statement that happens in the left choice, however, is the test $?x - o > 5$, which the system run has to pass to continue successfully. In particular, if $x - o > 5$ is indeed *true* in the current state, then the system passes that test $?x - o > 5$ and the

[2]In a more general context, tests are also known as *challenges* [Pla13].

execution proceeds to after the sequential composition (;) to run $a := a + 1$. If $x - o > 5$ is *false* in the current state, however, the system fails the test $?x - o > 5$ and that run is aborted and discarded. The right option to brake is always available, because it does not involve any tests to pass.

Note 4 (Discarding failed runs)**.** *System runs that fail tests are discarded and not considered any further. It is as if those failed system execution attempts had never happened. Yet, other execution paths may be successful. You can imagine finding them by backtracking the choices in the system run and taking alternative choices instead.*

There are always two choices when running (7). Yet, which ones run successfully depends on the current state. If the current state is at a far distance from the obstacle ($x - o > 5$), then both options of accelerating and braking will be possible. Otherwise, only the braking choice runs without being discarded because of failing a test.

Comparing (7) with (5), we see that (7) is a faithful abstraction of the more complicated (5), because all runs of (5) can be mimicked by (7). Yet, unlike (6), the improved HP (7) retains the critical information that acceleration is only allowed by (5) at sufficient distance $x - o > 5$. Unlike (5), (7) does not restrict the cases where acceleration can be chosen to those that also satisfy $v < 4 \land S$. Hence, (7) is more permissive than (5). But (7) is also simpler and only contains crucial information about the controller. Hence, (7) is a more abstract faithful model of (5) that retains the relevant detail. Studying the abstract (7) instead of the more concrete (5) has the advantage that only relevant details need to be understood while irrelevant aspects can be ignored. It also has the additional advantage that a safety analysis of the more abstract (7), which allows lots of behavior, will imply safety of the special concrete case (5) but also implies safety of other implementations of (7). For example, replacing S by a different condition in (5) still gives a special case of (7). So if all behavior of (7) is safe, all behavior of that different replacement will also already be safe. With a single verification result about a more general, more abstract system, we can obtain verification for a whole class of systems. This important phenomenon will be investigated in more detail in later parts of the course.

Of course, which details are relevant and which ones can be simplified depends on the analysis question at hand, a question that we will be better equipped to answer in a later lecture. For now, suffice it to say that (7) has the relevant level of abstraction for our purposes.

6 Repetitions in Hybrid Programs

The hybrid programs above were interesting, but only allowed the controller to choose what action to take at most once. All controllers so far inspected the state in a test or in an if-then-else condition and then chose what to do once, just to let physics take control by following a differential equation. That makes for rather short-lived controllers. They have a job only once in their lives. And most decisions they reach may end up being bad ones. Say, one of those controllers, e.g. (7), inspects the state and finds it still okay to accelerate. If it chooses $a := a + 1$ and then lets physics move in the differential equation $x'' = a$, there will probably come a time at which acceleration is no longer such a great idea. But the controller of (7) has no way to change its mind, because he has no more choices and so no control anymore.

If the controller of (7) is supposed to be able to make a second control choice later after physics has followed the differential equation for a while, then (7) can be sequentially composed with itself:

$$\begin{aligned}
&\big((?x - o > 5; a := a + 1) \cup a := -b\big);\\
&x' = v, v' = a;\\
&\big((?x - o > 5; a := a + 1) \cup a := -b\big);\\
&x' = v, v' = a
\end{aligned} \tag{8}$$

In (8), the cyber controller can first choose to accelerate or brake (depending on whether $x - o > 5$), then physics evolves along differential equation $x'' = a$ for some while, then the controller can again choose whether to accelerate or brake (depending on whether $x - o > 5$ holds in the state reached then), and finally physics again evolves along $x'' = a$.

For a controller that is supposed to be allowed to have a third control choice, replication would again help:

$$\begin{aligned}
&\big((?x - o > 5; a := a + 1) \cup a := -b\big);\\
&x' = v, v' = a;\\
&\big((?x - o > 5; a := a + 1) \cup a := -b\big);\\
&x' = v, v' = a;\\
&\big((?x - o > 5; a := a + 1) \cup a := -b\big);\\
&x' = v, v' = a
\end{aligned} \tag{9}$$

But this is neither a particularly concise nor a particularly useful modeling style. What if a controller could need 10 control decisions or 100? Or what if there is no way of telling ahead of time how many control decisions the cyber part will have to take to reach its goal? Think of how many control decisions you might need when driving in a car from the East Coast to the West Coast. Do you know that ahead of time? Even if you do, do you want to model a system by explicitly replicating its controller that often?

Note 5 (Repetition)**.** *As a more concise and more general way of describing repeated control choices, hybrid programs allow for the repetition operator* **, which works like the* * *in regular expressions, except that it applies to a hybrid program* α *as in* α^**. It repeats* α *any number* $n \in \mathbb{N}$ *of times, by a nondeterministic choice.*

Thus, a way of summarizing (7), (8), (9) and the infinitely many more n-fold replications of (7) for any $n \in \mathbb{N}$, is by using a repetition operator:

$$\Big(\big((?x - o > 5; a := a + 1) \cup a := -b\big);\\ x' = v, v' = a\Big)^* \tag{10}$$

This HP can repeat (7) any number of times.

But how often does a nondeterministic repetition like (10) repeat then? That choice is again nondeterministic.

Note 6 (Nondeterministic *)**.** *Repetition (*) is* nondeterministic. *That is, α^* can repeat α any number ($n \in \mathbb{N}$) of times and the choice how often to run α is* nondeterministic, *i.e. there is no prior way of telling how often α will be repeated.*

Yet, every time the loop in (10) is run, how long does its continuous evolution take? Or, actually, even in the loop-free (8), how long does the first $x'' = a$ take before the controller has its second choice? How long did the continuous evolution take in (7) in the first place?

There is a choice in following a differential equation. Even if the solution of the differential equation is unique (cf. lecture 2), it is still a matter of choice how long to follow that solution. The choice is, as always in hybrid programs, nondeterministic.

Note 7 (Nondeterministic $x' = \theta$)**.** *The duration of evolution of a differential equation ($x' = \theta \,\&\, H$) is* nondeterministic *(except that the evolution can never be so long that the state leaves H). That is, $x' = \theta \,\&\, H$ can follow the solution of $x' = \theta$ any amount of time ($0 \leq r \in \mathbb{R}$) of times and the choice how long to follow $x' = \theta$ is* nondeterministic, *i.e. there is no prior way of telling how often $x' = \theta$ will be repeated (except that it can never leave H).*

7 Syntax of Hybrid Programs

With the motivation above, we formally define hybrid programs [Pla12a, Pla10].

Definition 1 (Hybrid program)**.** HPs are defined by the following grammar (α, β are HPs, x a variable, θ a term possibly containing x, and H a formula of first-order logic of real arithmetic):

$$\alpha, \beta \ ::= \ x := \theta \mid ?H \mid x' = \theta \,\&\, H \mid \alpha \cup \beta \mid \alpha; \beta \mid \alpha^*$$

The first three cases are called atomic HPs, the last three compound. The *test* action $?H$ is used to define conditions. Its effect is that of a *no-op* if the formula H is true in the current state; otherwise, like *abort*, it allows no transitions. That is, if the test succeeds because formula H holds in the current state, then the state does not change, and the system execution continues normally. If the test fails because formula H does not hold in the current state, then the system execution cannot continue, is cut off, and not considered any further.

Nondeterministic choice $\alpha \cup \beta$, sequential composition $\alpha; \beta$, and nondeterministic repetition α^* of programs are as in regular expressions but generalized to a semantics in hybrid systems. *Nondeterministic choice* $\alpha \cup \beta$ expresses behavioral alternatives between the runs of α and β. That is, the HP $\alpha \cup \beta$ can choose nondeterministically to follow the runs of HP α, or, instead, to follow the runs of HP β. The *sequential composition* $\alpha; \beta$ models that the HP β starts running after HP α has finished (β never starts if α does not terminate). In $\alpha; \beta$, the runs of α take effect first, until α terminates (if it does), and then β continues. Observe that, like repetitions, continuous evolutions within α can take more or less time, which causes uncountable nondeterminism. This nondeterminism occurs in hybrid systems, because they can operate in so many different ways, which is as such reflected in HPs. *Nondeterministic repetition* α^* is used to express that the HP α repeats any number of times, including zero times. When following α^*, the runs of HP α can be repeated over and over again, any nondeterministic number of times ($\geq$0).

Unary operators (including *) bind stronger than binary operators and let ; bind stronger than $\cup$, so $\alpha; \beta \cup \gamma \equiv (\alpha; \beta) \cup \gamma$ and $\alpha \cup \beta; \gamma \equiv \alpha \cup (\beta; \gamma)$. Further, $\alpha; \beta^* \equiv \alpha; (\beta^*)$.

8 Semantics of Hybrid Programs

HPs have a compositional semantics [Pla12b, Pla10, Pla08]. Their semantics is defined by a reachability relation. A *state* ν is a mapping from variables to $\mathbb{R}$. The set of states is denoted $\mathcal{S}$. The value of term θ in ν is denoted by $[\![\theta]\!]_\nu$. Recall that $\nu \models H$ denotes that first-order formula H is true in state ν (lecture 2).

Definition 2 (Transition semantics of HPs)**.** Each HP α is interpreted semantically as a binary reachability relation $\rho(\alpha) \subseteq \mathcal{S} \times \mathcal{S}$ over states, defined inductively by

1. $\rho(x := \theta) = \{(\nu, \omega) \ : \ \omega = \nu$ except that $[\![x]\!]_\omega = [\![\theta]\!]_\nu\}$
2. $\rho(?H) = \{(\nu, \nu) \ : \ \nu \models H\}$
3. $\rho(x' = \theta \,\&\, H) = \{(\varphi(0), \varphi(r)) \ : \ \varphi(t) \models x' = \theta$ and $\varphi(t) \models H$ for all $0 \leq t \leq r$ for a solution $\varphi : [0, r] \to \mathcal{S}$ of any duration $r\}$; i.e., with $\varphi(t)(x') \overset{\text{def}}{=} \frac{\mathsf{d}\varphi(\zeta)(x)}{\mathsf{d}\zeta}(t)$, φ solves the differential equation and satisfies H at all times, see lecture 2.
4. $\rho(\alpha \cup \beta) = \rho(\alpha) \cup \rho(\beta)$
5. $\rho(\alpha; \beta) = \rho(\beta) \circ \rho(\alpha) = \{(\nu, \omega) : (\nu, \mu) \in \rho(\alpha), (\mu, \omega) \in \rho(\beta)\}$
6. $\rho(\alpha^*) = \bigcup_{n \in \mathbb{N}} \rho(\alpha^n)$ with $\alpha^{n+1} \equiv \alpha^n; \alpha$ and $\alpha^0 \equiv ?true$.

For graphical illustrations of the transition semantics of hybrid programs and example dynamics, see Fig. 1. On the left of Fig. 1, we illustrate the generic shape of the transition structure $\rho(\alpha)$ for transitions along various cases of hybrid programs α from state ν to state ω. On the right of Fig. 1, we show examples of how the value of a variable x may evolve over time t when following the dynamics of the respective hybrid program α.

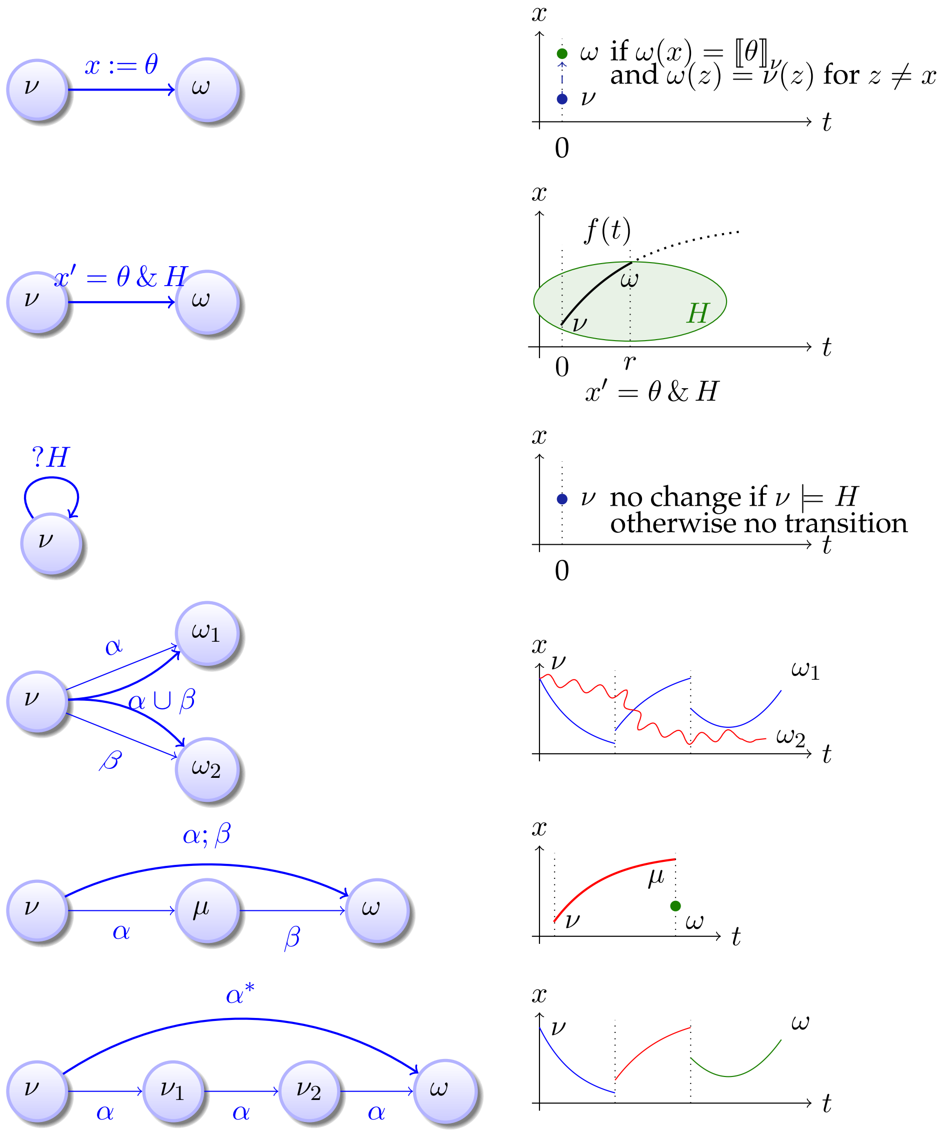

Figure 1: Transition semantics (left) and example dynamics (right) of hybrid programs

Exercises

Exercise 1. Consider your favorite programming language and discuss in what ways it introduces discrete change and discrete dynamics. Can it model all behavior that hybrid programs can describe? Can your programming language model all behavior that hybrid programs without differential equations can describe? How about the other way around?

Exercise 2. Consider the grammar of hybrid programs. The ; in hybrid programs is similar to the ; in Java and C0. If you look closely you will find a subtle difference. Identify the difference and explain why there is such a difference.

Exercise 3. Sect. 3 considered if-then-else statements for hybrid programs. But they no longer showed up in the grammar of hybrid programs. Is this a mistake?

References

[DBL12] *Proceedings of the 27th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25–28, 2012*. IEEE, 2012.

[Pla07] André Platzer. Differential dynamic logic for verifying parametric hybrid systems. In Nicola Olivetti, editor, *TABLEAUX*, volume 4548 of *LNCS*, pages 216–232. Springer, 2007. doi:10.1007/978-3-540-73099-6_17.

[Pla08] André Platzer. Differential dynamic logic for hybrid systems. *J. Autom. Reas.*, 41(2):143–189, 2008. doi:10.1007/s10817-008-9103-8.

[Pla10] André Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer, Heidelberg, 2010. doi:10.1007/978-3-642-14509-4.

[Pla12a] André Platzer. The complete proof theory of hybrid systems. In *LICS* [DBL12], pages 541–550. doi:10.1109/LICS.2012.64.

[Pla12b] André Platzer. Logics of dynamical systems. In *LICS* [DBL12], pages 13–24. doi:10.1109/LICS.2012.13.

[Pla13] André Platzer. A complete axiomatization of differential game logic for hybrid games. Technical Report CMU-CS-13-100R, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, January, Revised and extended in July 2013.