15-424: Foundations of Cyber-Physical Systems

Lecture Notes on Choice & Control

André Platzer

Carnegie Mellon University
Lecture 3

1 Introduction

In the Lecture 2 on Differential Equations & Domains, we have seen the beginning of cyber-physical systems, yet emphasized their continuous part in the form of differential equations $x' = f(x)$. The sole interface between continuous physical capabilities and cyber capabilities was by way of their evolution domain. The evolution domain Q in a continuous program $x' = f(x) \,\&\, Q$ 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 Q if it had kept going. Then 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, which is exactly the model we need to describe the hybrid system dynamics of cyber-physical systems. Hybrid programs combine discrete and continuous dynamics in a seamless way. This lecture is based on material on cyber-physical systems and hybrid programs [Pla12b, Pla10, Pla08, Pla07].

Continuous programs $x' = f(x) \,\&\, Q$ are very powerful for modeling continuous processes. They cannot—on their own—model discrete changes of variables, however[1], which is useful to understand the impact of computer decisions on cyber-physical systems. During the evolution along a differential equation, all variables change continuously in time, because the solution of a differential equation is (sufficiently) smooth.

[1]There is a much deeper sense [Pla12a] in which continuous dynamics and discrete dynamics have surprising similarities regardless. But even so, these similarities rest on the foundations of hybrid systems, which we need to understand first.

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?

There are many models for describing discrete change. Other computer science topics provide ample opportunity for seeing models of discrete change in action. CPSs combine cyber and physics, though. 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. So a programming language would give us a model of discrete computation. Of course, for programming an actual CPS, our programming language will ultimately have to involve physics. So, none of the conventional programming languages alone will work for CPS. But we have already seen continuous programs in the previous lecture for that very purpose. What's missing in continuous programs is a way to program the discrete and cyber aspects, which is exactly what the features of conventional programming languages provide. Which means that set course on an expedition to combine conventional discrete programming languages with the continuous program we discovered last time.

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 (also see Church-Turing thesis [Chu36, Tur37]). Yet even among all those conventional programming languages 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, culminating in the programming language of *hybrid programs* (HP).

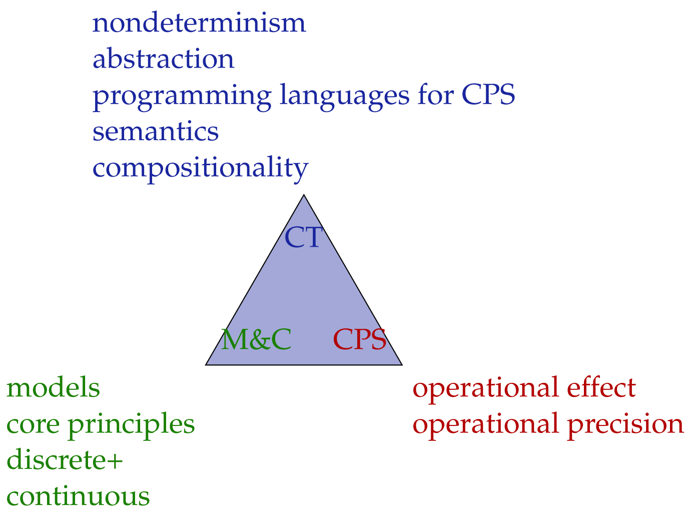
More information about choice and control can be found in [Pla10, Chapter 2.2,2.3].

The most important learning goals of this lecture are:

Modeling and Control: This lecture plays a crucial role in understanding and designing models of CPSs. We develop an understanding of the core principles behind CPS by studying how discrete and continuous dynamics are combined and interact to model cyber and physics, respectively. We see the first example of how to develop models and controls for a (simplistic) CPS. Even if subsequent lectures will blur the overly simplistic categorization of cyber=discrete versus physics=continuous, it is useful to equate them for now, because cyber and computation and decisions quickly lead to discrete dynamics while physics gives rise to continuous dynamics, naturally. In later lectures, we will discover that some physical phenomena are better modeled with discrete dynamics while some controller aspects have a manifestation in continuous dynamics.

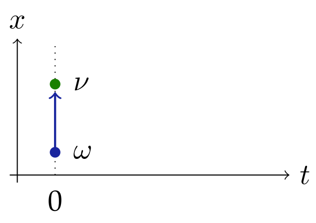
Computational Thinking: We introduce and study the important phenomenon of nondeterminism, which is crucial for developing faithful models of a CPS's environment and helpful for developing efficient models of the CPS itself. We emphasize the importance of abstraction, which is an essential modular organization principle in CPS as well as all other parts of computer science. We capture the core aspects of CPS in a programming language, the language of hybrid programs.

CPS Skills: We develop an intuition for the operational effects of CPS. And we will develop an understanding for the semantics of the programming language of hybrid programs, which is the CPS model that this course is based on.



2 Discrete Programs and Sequential Composition

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



This gives us a discrete model of change, $x := e$, in addition to the continuous model of change, $x' = f(x) \,\&\, Q$ from the Lecture 2 on Differential Equations & Domains. Now, we can model systems that are *either* discrete *or* continuous. Yet, how can we model proper CPS that combine cyber and physics with one another and that, thus, simultaneously combine discrete and continuous dynamics? We need such hybrid behavior every time a system has both continuous dynamics (such as the continuous motion of a car down the street) and discrete dynamics (such as shifting gears).

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 the cyber part first does something and physics then follows. Such a behavior corresponds to a sequential composition $(\alpha; \beta)$ in which first the HP α on the left of the sequential composition operator ; runs and, when it's done, the HP β on the right of ; runs. For example, the following HP[2]

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

will first let cyber perform a discrete change of setting a to $a + 1$ and then let physics follow the differential equation $x'' = a$,[3] which describes accelerated motion of point x along a straight line. The overall effect is that cyber increases the value of variable a and physics then lets x evolve with acceleration a (and increases velocity v continuously 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 fixed 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 minor difference in that programming languages like Java and C0 expect more ; than hybrid programs, for example at the end of the last statement. This difference is inconsequential, and a common treat of mathematical programming languages.

The HP in (1) executes control (it sets the acceleration for physics), but it has very little choice. Actually no choice on what happes at all. So only if the CPS is very lucky will an increase in acceleration be the right action to remain safe forever. Quite likely, the robot will have to change its mind ultimately, which is what we will investigate next.

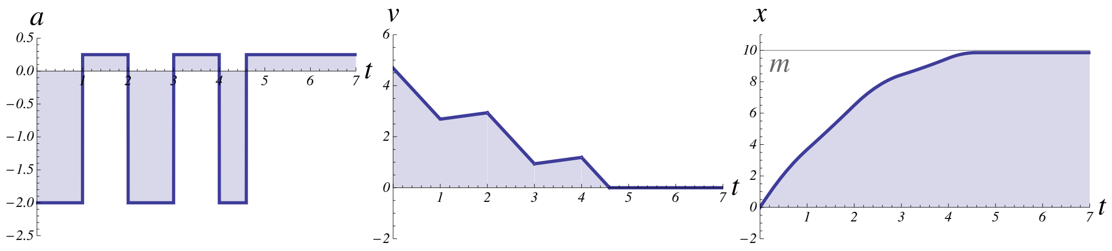


Figure 1: Acceleration a **(left)**, velocity v **(middle)**, and position x **(right)** change over time t, with acceleration changing discretely at instants of time while velocity and position change continuously over time.

But first observe that the constructs we saw so far, assignments, sequential compositions, and differential equations already suffice to exhibit typical hybrid systems dy-

[2] Note that the parentheses around the differential equation are redundant and will often be left out in the lecture notes or in scientific papers. HP (1), for example, would be written $a := a + 1;\ \{x' = v, v' = a\}$. The KeYmaera X theorem prover that will be using in this course insist on more brackets, however, and, in fact, use braces for differential equations and programs: `a:=a+1; {x'=v,v'=a}`.

[3] We will frequently use $x'' = a$ as an abbreviation for $x' = v, v' = a$, even if x'' is not officially permitted in KeYmaera X.

namics. The behavior shown in Fig. 1 could be exhibited by the hybrid program:

$$\begin{aligned}&a := -2;\ \{x' = v, v' = a\};\\&a := 0.25;\ \{x' = v, v' = a\};\\&a := -2;\ \{x' = v, v' = a\};\\&a := 0.25;\ \{x' = v, v' = a\};\\&a := -2;\ \{x' = v, v' = a\};\\&a := 0.25;\ \{x' = v, v' = a\}\end{aligned}$$

Can you already spot a question that comes up about how exactly we run this program? We will postpone the formulation and answer to this question to Sect. 6.

3 Decisions in Hybrid Programs

In general, a CPS will have to check conditions on the state to see which action to take. Otherwise it could not possibly be safe and, quite likely, will also not be able to take the right turns to get to its goal. One way of programming these conditions 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 than 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 and is often a distraction from the really important aspects of a system. 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 $x-o>5\land v<4$ holds, unlike (5), which also 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 to the original system? What we want this model to do is characterize that the controller may either increase acceleration by 1 or brake. And we want that all acceleration certainly only happens when $x-o>5$. But the model should make less commitment than (3) about the precise circumstances under which braking is chosen. After all, braking may sometimes just be the right thing to do. 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). The rationale is ultimately that if a system with more behavior is safe, the actual implementation will be safe as well, because it will only ever exercise some of the verified behavior. And the extra behavior in the system might, in fact, even happen in reality whenever there are minor lags or discrepancies. So it is good to have the extra assurance that some flexibility in the execution of the system will not break its safety guarantees.

Note 2 (Abstraction)**.** *Successful CPS models often include only the relevant aspects of the system 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, probably 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 throughout your entire career.*

Let us take the development of this model step by step. The first feature that the controller of the 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 that the first statement (before the ;) runs, which is a choice ($\cup$) between whether to run $a := a + 1$ or whether to run $a := -b$. That is, the choice is whether to increase acceleration a by 1 or whether to reset a to $-b$ for braking. After this choice (i.e. after the ; sequential composition operator), the system follows the usual differential equation $x'' = a$ describing accelerated motion along a line.

Now, wait. There was a choice. Who choses? How is the choice resolved?

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. Both outcomes are perfectly possible.*

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, and, thus, not a very suitable abstraction.

5 Tests in Hybrid Programs

In order to build a faithful yet not overly permissive model of (5), we need to restrict the permitted choices in (6) so that there's flexibility but only so much 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* $?Q$ is a statement that checks the truth-value of a first-order formula Q of real arithmetic in the current state. If Q holds in the current state, then the test passes, nothing happens, yet the HP continues to run normally. If, instead, Q does not hold in the current state, then the test fails, and the system execution is aborted and discarded. That is, when ω is the current state, then $?Q$ runs successfully without changing the state when $\omega \in [\![Q]\!]$. Otherwise, i.e. if $\omega \notin [\![Q]\!]$, the run of $?Q$ is aborted and not considered any further, because it did not play by the rules of the system.

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 in order to be able 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 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, because that run did not play by the rules of the system. It is as if those failed system execution attempts had never happened. Yet, even if one execution attempt fails, other execution paths may still be successful. Operationally, you can imagine finding them by backtracking through all 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 indeed be possible and can be run successfully. Otherwise, only the braking choice runs without being discarded because of a failing 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 the intermediate guess (6), the improved HP (7) still 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 rather than just one particular system. 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.

Note 5 (Broader significance of nondeterminism)**.** *Nondeterminism comes up in the above cases for reasons of abstraction and for focusing the system model on the most critical aspects of the system while suppressing irrelevant detail. This is an important reason for introducing nondeterminism in system models, but there are other important reasons as well. Whenever a system includes models of its environment, nondeterministic models are often a crucial idea, because there is often just a partial understanding of what the environment will do. A car controller for example, will not always know for sure what other cars in its environment will do, exactly, so that nondeterministic models are the only faithful representations.*

Note the notational convention that sequential composition ; binds stronger than nondeterministic choice $\cup$ so we can leave some parentheses out without changing (7):

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

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 subsequently 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 it 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 simply 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, copy&paste 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 6 (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 star operator 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, including 0, by a nondeterministic choice.*

Thus, the programmatic 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 instead:

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

This HP can repeat (7) any number of times (0,1,2,3,...). Of course, it would not be very meaningful to repeat a loop half a time or minus 5 times, so the repetition count $n \in \mathbb{N}$ has to be some natural number.

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

Note 7 (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, hold on, every time the loop in (10) is run, how long does the continuous evolution along $\{x' = v, v' = a\}$ in that loop iteration take? Or, actually, even in the loop-free (8), how long does the first $x'' = a$ take before the controller has its second control choice? How long did the continuous evolution take in (7) in the first place?

There is a choice even in following a differential equation! However deterministic the solution of the differential equation itself may be. Even if the solution of the differential equation is unique (which it is in sufficiently smooth cases that we consider 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 8 (Nondeterministic $x' = f(x)$)**.** *The duration of evolution of a differential equation ($x' = f(x) \,\&\, Q$) is* nondeterministic *(except that the evolution can never be so long that the state leaves Q). That is, $x' = f(x) \,\&\, Q$ can follow the solution of $x' = f(x)$ any amount of time ($0 \leq r \in \mathbb{R}$) of times and the choice how long to follow $x' = f(x)$ is* nondeterministic, *i.e. there is no prior way of telling how often $x' = f(x)$ will be repeated (except that it can never leave Q).*

7 Syntax of Hybrid Programs

With the motivation above, we formally define the programming language of hybrid programs [Pla12a, Pla10], in which all of the operators that we motivated above are allowed.

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

$$\alpha, \beta \ ::= \ x := e \mid ?Q \mid x' = f(x) \,\&\, Q \mid \alpha \cup \beta \mid \alpha ; \beta \mid \alpha^*$$

The first three cases are called *atomic HPs*, the last three *compound HPs* because they are built out of smaller HPs. The *test* action $?Q$ is used to define conditions. Its effect is that of a *no-op* if the formula Q is true in the current state; otherwise, like *abort*, it allows no transitions. That is, if the test succeeds because formula Q holds in the current state, then the state does not change (it was only a test), and the system execution continues normally. If the test fails because formula Q does not hold in the current state, however, then the system execution cannot continue, is cut off, and not considered any further since it is a failed execution attempt that did not play by the rules of the HP.

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 ; binds 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

After having developed a syntax for CPS and an operational intuition for its effects, we seek operational precision in its effects. That is, we will pursue one important leg of computational thinking and give an unambiguous meaning to all operators of HPs. We will do this in pursuit of the realization that the only way to be precise about an analysis of CPS is to first be precise about the meaning of the models of CPS. Furthermore, we will leverage another important leg of computational thinking rooted in logic by exploiting that the right way of understanding something is to understand it compositionally as a function of its pieces. So we will give meaning to hybrid programs by giving a meaning to each of its operators. Thereby, a meaning of a large HP is merely a function of the meaning of its pieces. This is the style of denotational semantics due to Scott and Stratchey [SS71].

There is more than one way to define the meaning of a program, including defining a denotational semantics, an operational semantics, a structural operational semantics, or an axiomatic semantics. For our purposes, what is most relevant is how a hybrid program changes the state of the system. Consequently, the semantics of HPs is based on which final states are reachable from which initial state. It considers which (final) state ν is reachable by running a HP α from an (initial) state ω. Semantical models that expose more detail, e.g., about the internal states during the run of an HP are possible [Pla10, Chapter 4] but can be ignored for most models.

Recall that a *state* ω is a mapping from variables to $\mathbb{R}$. The set of states is denoted $\mathcal{S}$.

The meaning of an HP α is given by a reachability relation $[\![\alpha]\!] \subseteq \mathcal{S} \times \mathcal{S}$ on states. That is, $(\omega, \nu) \in [\![\alpha]\!]$ means that final state ν is reachable from initial state ω by running HP α. From any initial state ω, there might be many states ν that are reachable because the HP α may involve nondeterministic choices, repetitions or differential equations, so there may be many different ν for which $(\omega, \nu) \in [\![\alpha]\!]$. Form other initial states ω, there might be no reachable states ν at all for which $(\omega, \nu) \in [\![\alpha]\!]$. So $[\![\alpha]\!]$ is a proper relation, not a function.

HPs have a compositional semantics [Pla12b, Pla10, Pla08]. Recall that the value of term e in ω is denoted by $[\![e]\!]\omega$ and that $\mathcal{S}$ denotes the set of all states. Further, $\omega \in [\![Q]\!]$ denotes that first-order formula Q is true in state ω (Lecture 2 on Differential Equations & Domains). The semantics of an HP α is defined by its reachability relation $[\![\alpha]\!] \subseteq \mathcal{S} \times \mathcal{S}$.

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

1. $[\![x := e]\!] = \{(\omega, \nu) \ : \ \nu = \omega$ except that $[\![x]\!]\nu = [\![e]\!]\omega\}$
   That is, final state ν differs from initial state ω only in its interpretation of the variable x, which ν changes to the value that the right-hand side e has in the initial state ν.

2. $[\![?Q]\!] = \{(\omega, \omega) \ : \ \omega \in [\![Q]\!]\}$
   That is, the final state ω is the same as the initial state ω (no change) but there only is such a self-loop transition if test formula Q holds in ω, otherwise no transition is possible at all and the system is stuck because of a failed test.

3. $[\![x' = f(x) \,\&\, Q]\!] = \{(\varphi(0), \varphi(r)) \ : \ \varphi(\zeta) \models x' = f(x)$ and $\varphi(\zeta) \models Q$ for all $0 \leq \zeta \leq r$ for a solution $\varphi : [0, r] \to \mathcal{S}$ of any duration $r\}$
   That is, the final state $\varphi(r)$ is connected to the initial state $\varphi(0)$ by a continuous function of some duration $r \geq 0$ that solves the differential equation and satisfies Q at all times, when interpreting $\varphi(\zeta)(x') \overset{\text{def}}{=} \frac{d\varphi(t)(x)}{dt}(\zeta)$ as the derivative of the value of x over time at time ζ, see Lecture 2.

4. $[\![\alpha \cup \beta]\!] = [\![\alpha]\!] \cup [\![\beta]\!]$
   That is, $\alpha \cup \beta$ can do any of the transitions that α can do as well as any of the transitions that β is capable of.

5. $[\![\alpha; \beta]\!] = [\![\alpha]\!] \circ [\![\beta]\!] = \{(\omega, \nu) : (\omega, \mu) \in [\![\alpha]\!], (\mu, \nu) \in [\![\beta]\!]\}$
   That is, the meaning of $\alpha; \beta$ is the composition[a] $[\![\alpha]\!] \circ [\![\beta]\!]$ of relation $[\![\beta]\!]$ after $[\![\alpha]\!]$. Thus, $\alpha; \beta$ can do any transitions that go through any intermediate state μ to which α can make a transition from the initial state ω and from which β can make a transition to the final state ν.

6. $[\![\alpha^*]\!] = \bigcup_{n \in \mathbb{N}} [\![\alpha^n]\!]$ with $\alpha^{n+1} \equiv \alpha^n; \alpha$ and $\alpha^0 \equiv ?true$.
   That is, α^* can repeat α any number of times, i.e., for any $n \in \mathbb{N}$, α^* can act like the n-fold sequential composition α^n would.

[a]The notational convention for composition of relations is flipped compared to the composition of functions. For functions f and g, the function $f \circ g$ is the composition f after g that maps x to $f(g(x))$. For relations R and T, the relation $R \circ T$ is the composition of T after R, so first follow relation R to an intermediate state and then follow relation T to the final state.

To keep things simple, the above definition uses simplifying abbreviations for differential equations. Lecture 2 provides full detail also of the definition for differential equation systems rather than single differential equations.

For graphical illustrations of the transition semantics of hybrid programs and example dynamics, see Fig. 2. The left of Fig. 2 illustrates the generic shape of the transition structure $[\![\alpha]\!]$ for transitions along various cases of hybrid programs α from state ω to

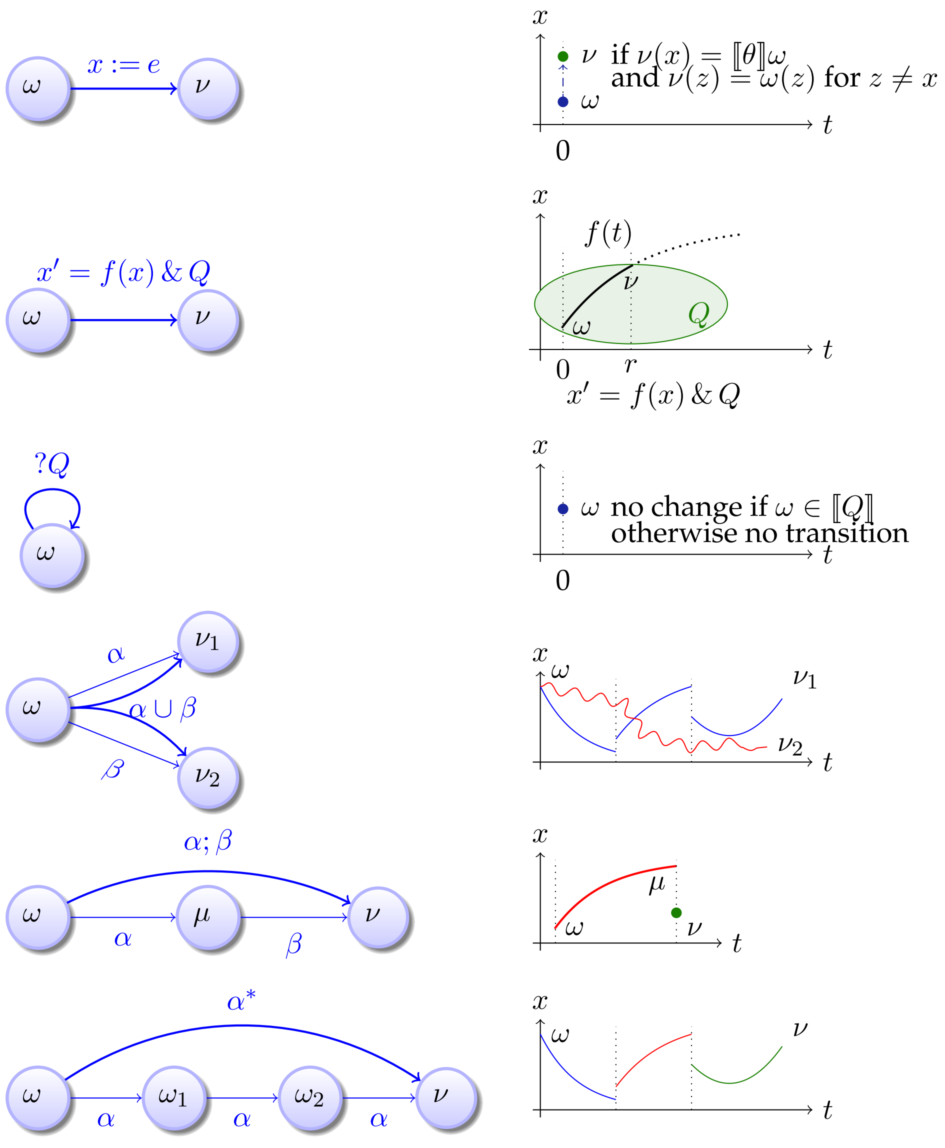


Figure 2: Transition semantics (left) and example dynamics (right) of hybrid programs

state ν. The right of Fig. 2 shows examples of how the value of a variable x may evolve over time t when following the dynamics of the respective hybrid program α.

Now when α denotes the HP in (8), its semantics $[\![\alpha]\!]$ is a relation on states connecting the initial to the final state along the differential equation with two control decisions according to the nondeterministic choice, one at the beginning and one after following the first differential equation. How long that is, exactly? Well, that's nondeterministic, because the semantics of differential equations is such that any final state after any permitted duration is reachable from a given initial state. So the duration for the first differential equation in (8) could have been one second or two or 424 or half a second or zero or any other nonnegative real number.

If we change the HP around and consider the following modification instead:

$$\begin{aligned}
&?x - o > 5; a := a + 1;\\
&\{x' = v, v' = a\};\\
&?x - o > 5; a := a + 1;\\
&\{x' = v, v' = a\}
\end{aligned} \tag{11}$$

Then some behavior that was still possible in (8) is no longer possible for (11). Let β denote the HP in (11), then the semantics $[\![\beta]\!]$ of β now only includes relations between initial and final states which can be reached by acceleration choices (because there are no more braking choices in β). In particular, however, note that the duration of the first differential equation in (11) may suddenly be bounded, because if x keeps on accelerating for too long during the first differential equation, the intermediate state reached then will violate the test $?x - o > 5$, which, according to the semantics of tests, will fail and be discarded. Of course, if x accelerates for too long, it will surely ultimately violate the condition that its position will be at least 5 in front of the obstacle o.

9 Summary

This lecture introduced hybrid programs as a model for cyber-physical systems, summarized in Note 11. Hybrid programs combine differential equations with conventional program constructs and discrete assignments. The programming language of hybrid programs embraces nondeterminism as a first-class citizen and features differential equations that can be combined to form hybrid systems using the compositional operators of hybrid programs.

Note 11 (Statements and effects of hybrid programs (HPs)).

HP Notation	*Operation*	*Effect*
$x := e$	*discrete assignment*	*assigns term e to variable x*
$x' = f(x) \,\&\, Q$	*continuous evolution*	*differential equations for x with term $f(x)$ within first-order constraint Q (evolution domain)*
$?Q$	*state test / check*	*test first-order formula Q at current state*
$\alpha;\ \beta$	*seq. composition*	*HP β starts after HP α finishes*
$\alpha \cup \beta$	*nondet. choice*	*choice between alternatives HP α or HP β*
α^*	*nondet. repetition*	*repeats HP α n-times for any $n \in \mathbb{N}$*

Exercises

Exercise 1. The semantics of hybrid programs (Def. 2) requires evolution domain constraints Q to hold always throughout a continuous evolution. What exactly happens if the system starts in a state where Q does not hold to begin with?

Exercise 2. 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 3. 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 4. 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?

Exercise 5. The semantics of hybrid programs (Def. 2) is defined as a transition relation $[\![\alpha]\!] \subseteq \mathcal{S} \times \mathcal{S}$ on states. Define an equivalent semantics based on functions $R(\alpha) : \mathcal{S} \to 2^{\mathcal{S}}$ from the initial state to the set of all final states, where $2^{\mathcal{S}}$ denotes the powerset of $\mathcal{S}$, i.e. the set of all subsets of $\mathcal{S}$. Define this set-valued semantics $R(\alpha)$ without referring to the transition relation semantics $[\![\alpha]\!]$. Likewise, define an equivalent semantics based on functions $\varsigma(\alpha) : 2^{\mathcal{S}} \to 2^{\mathcal{S}}$ from the set of all initial states to the set of all final states.

References

[Chu36] Alonzo Church. A note on the Entscheidungsproblem. *J. Symb. Log.*, 1(1):40–41, 1936.

[LIC12] *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 [LIC12], pages 541–550. doi:10.1109/LICS.2012.64.

[Pla12b] André Platzer. Logics of dynamical systems. In LICS [LIC12], pages 13–24. doi:10.1109/LICS.2012.13.

[SS71] Dana Scott and Christopher Strachey. Toward a mathematical semantics for computer languages? Technical Report PRG-6, Oxford Programming Research Group, 1971.

[Tur37] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society 2*, 42:230–265, 1937.