André Platzer

Lecture Notes on Foundations of Cyber-Physical Systems

15-424/624/824 Foundations of Cyber-Physical Systems

Chapter 3
Choice & Control

Synposis This chapter develops the central dynamical systems model for describing the behavior of cyber-physical systems with a programming language. It complements the previous understanding of continuous dynamics with an understanding of the discrete dynamics caused by choices and controls in cyber-physical systems. The chapter interfaces the continuous dynamics of differential equations with the discrete dynamics of conventional computer programs by directly integrating differential equations with discrete programming languages. This leverages well-established programming language constructs around elementary discrete and continuous statements to obtain *hybrid programs* as a core programming language for cyber-physical systems. In addition to embracing differential equations, semantical generalizations to mathematical reals as well as operators for nondeterminism are important to make hybrid programs appropriate for cyber-physical systems.

3.1 Introduction

Chapter 2 saw the beginning of cyber-physical systems, yet emphasized only 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 part take control? How do we describe what the cyber elements compute afterwards? What descriptions explain how cyber interacts with physics?

An overall understanding of a CPS ultimately requires an understanding of the joint model with both its discrete dynamics and its continuous dynamics. It takes both to understand, for example, what effect a discrete car controller has, via its engine and steering actuators, on the continuous physical motion of a car down the road. Continuous programs are powerful for modeling continuous processes, such as continuous motion. They cannot—on their own—model discrete changes of variables, however.[1] Such discrete state change helps understand the impact of computer decisions on cyber-physical systems, in which computation decides to, say, stop speeding up and apply the brakes instead. During the evolution along a differential equation, such as $x' = v, v' = a$ for accelerated motion along a straight line, all variables change continuously over time, because the solution of a differential equation is (sufficiently) smooth. Discontinuous change of variables, such as a change of acceleration from $a = 2$ to $a = -6$, instead, needs a discrete change of state resulting from how computers compute decisions one step at a time. What could be a model for describing such discrete changes in a system?

Discrete change can be described by different models. The most prominent ones are conventional programming languages, in which everything takes effect one discrete step at a time, just like computer processors operate one clock cycle at a time.

CPSs combine cyber and physics, though. In CPS, we do not program computers, but program cyber-physical systems instead. We program the computers that control the physics, which requires programming languages for CPSs to involve physics, and integrate differential equations seamlessly with discrete computer operations. The basic idea is that its discrete statements are executed by a computer processor, while its continuous statements are handled by the physical elements, such as wheels, engines, or brakes. CPS programs need a mix of both, though, to accurately describe the combined discrete and continuous dynamics.

Does it matter which discrete programming language we choose to enrich with the continuous statements from Chap. 2? It could be argued that the hybrid aspects are more important for CPS than the discrete language. After all, there are many conventional programming languages that are Turing-equivalent, i.e. that compute the same functions [2, 7, 23]. Yet there are numerous significant differences even among discrete programming languages that make some more desirable than others [5]. For the particular purposes of CPS, we will identify additional desired features. We will develop what we need as we go, culminating in the programming language of *hybrid programs* [13–18], which plays a fundamental role in this book.

Other areas such as automata theory and the theory of formal languages [7] or Petri nets [12] also provide models of discrete change. There are ways of augmenting these models with differential equations as well [1, 3, 11]. But programming languages are uniquely positioned to extend their virtues of built-in compositionality. Just as the meaning and effect of a conventional program is a function of its pieces, the meaning and operation of a hybrid program is also a function of its parts.

The most important learning goals of this chapter are:

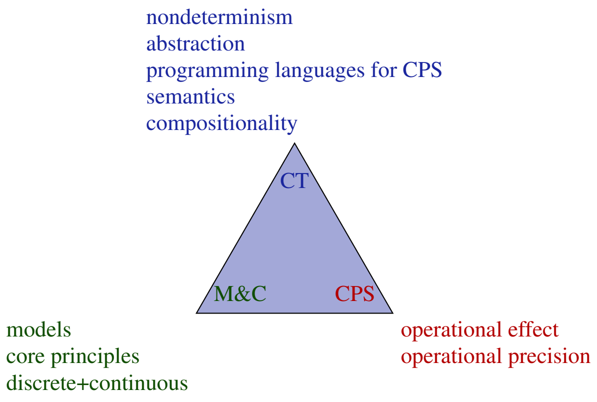
Modeling and Control: This chapter plays a crucial role in understanding and designing models of CPSs. We develop an understanding of the core principles behind CPS by studying how discrete and continuous dynamics are combined

[1] There is a much deeper sense [17] 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.

and interact to model cyber and physics, respectively. We see the first example of how to develop models and controls for a simple CPS. Even if subsequent chapters will blur the overly simplistic categorization of cyber=discrete versus physics=continuous, it is useful to equate them for now, because cyber, computation, and decisions quickly lead to discrete dynamics, while physics naturally gives rise to continuous dynamics. Later chapters will discover that some physical phenomena are better modeled with discrete dynamics, while some controller aspects also 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 effective 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 the programming 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 textbook is based on.



3.2 A Gradual Introduction to Hybrid Programs

This section gradually introduces the operations that hybrid programs provide one step at a time. Its emphasis is on their motivation and an intuitive development before subsequent sections provide a comprehensive view.

3.2.1 Discrete Change in Programs

Discrete change immediately 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 by evaluating the term e and assigning the result to the 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 an instantaneous discrete jump in the value of x.

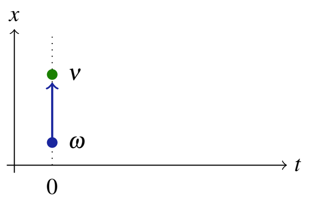


Fig. 3.1 An illustration of the behavior of an instantaneous discrete change at time = 0

This gives us a discrete model of change, $x := e$, in addition to the continuous model of change, $x' = f(x) \,\&\, Q$, from the Chap. 2. We can now 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) in addition to discrete dynamics (such as shifting gears).

3.2.2 Compositions of Programs

One way how cyber and physics can interact is if a computer provides input to physics. Physics may mention variables like a for acceleration and the 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 operator ; runs. The following HP[2]

$$a := a+1;\ \{x' = v, v' = a\} \tag{3.1}$$

[2] Note that the parentheses around the differential equation are redundant and will often be left out in the textbook or in scientific papers. HP (3.1) would be written $a := a+1;\ x' = v, v' = a$. Round parentheses are often used in theoretical developments, while braces are useful to disambiguate in bigger CPS applications.

will first let cyber perform a discrete change of setting acceleration variable a to $a+1$ and then let physics follow the differential equation[3] $x'' = a$, which describes accelerated motion of the point x along a straight line. The overall effect is that cyber instantly increases the value of acceleration variable a and physics then lets x evolve continuously with that acceleration a (increasing velocity v continuously with derivative a). HP (3.1) models a situation where the desired acceleration is commanded once to increase and the robot then moves with that fixed acceleration; see Fig. 3.2. The sequential composition operator (;) has the same effect that it has in programming languages like Java. It separates statements that are to be executed sequentially one after the other. If you look closely, you will find a minor subtle difference, because programming languages like Java or C expect ; at the end of every statement, not just in between sequentially composed statements. This syntactic difference is inconsequential, and a common treat of mathematical programming languages.

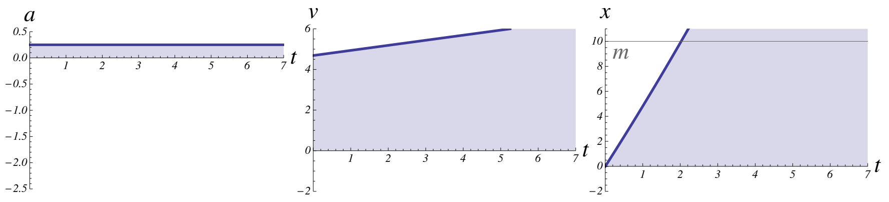


Fig. 3.2 Fixed acceleration a **(left)**, velocity v **(middle)**, and position x **(right)** change over time t

The HP in (3.1) 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 forever. Quite likely, the robot will have to change its mind ultimately, which is what we investigate next.

But first observe that the constructs we saw so far, assignments, sequential compositions, and differential equations already suffice to exhibit typical hybrid systems dynamics. The behavior shown in Fig. 3.3 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. 3.2.6.

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

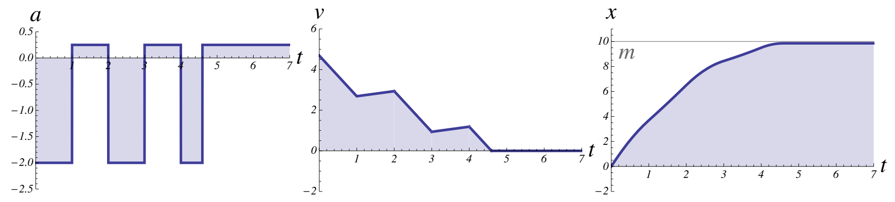


Fig. 3.3 Acceleration a **(left)**, velocity v **(middle)**, and position x **(right)** change over time t, with a piecewise constant acceleration changing discretely at instants of time while velocity and position change continuously over time.

3.2.3 Decisions in Hybrid Programs

In general, a CPS will have to check conditions on the state to see which action to take. Otherwise the CPS could not possibly be safe and, quite likely, will also not take the right turns to get to its goal. One way of programming these conditions is the use of an if-then-else statement, as in classical discrete programs.

$$\begin{aligned}&\mathsf{if}(v < 4)\, a := a + 1\ \mathsf{else}\ a := -b;\\&\{x' = v, v' = a\}\end{aligned} \tag{3.2}$$

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 in the first line has run to completion, the HP will again evolve x continuously with acceleration a along a differential equation in the second line.

The HP (3.2) 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 sufficient distance $x - m$ to an obstacle m from the robot's position x:

$$\begin{aligned}&\mathsf{if}(x - m > s)\, a := a + 1\ \mathsf{else}\ a := -b;\\&\{x' = v, v' = a\}\end{aligned} \tag{3.3}$$

Whether that is safe depends on the choice of the required safety distance s. Controllers could also take both distance *and* velocity into account for the decision:

$$\begin{aligned}&\mathsf{if}(x - m > s \land v < 4)\, a := a + 1\ \mathsf{else}\ a := -b;\\&\{x' = v, v' = a\}\end{aligned} \tag{3.4}$$

Note 9 (Iterative design) *To design serious controllers, you will usually develop a series of increasingly more intelligent controllers for systems that face increasingly challenging environments. Designing controllers for robots or other CPSs 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. If a controller is not even safe under simple circumstances (for example when it only knows how to brake), it will not be safe in more complex cases either.*

3.2.4 Choices in Hybrid Programs

A common feature of CPS models is that they often include only some but not all detail about the system. 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 (3.4) might have the following shape, with some further formula S as an extra condition for checking whether to actually accelerate based on battery efficiency or secondary considerations:

$$\begin{aligned}&\mathsf{if}(x-m>s\land v<4\land S)\,a:=a+1\,\mathsf{else}\,a:=-b;\\&\{x'=v,v'=a\}\end{aligned}\tag{3.5}$$

Consequently, (3.4) is not actually a faithful model for (3.5), because (3.4) insists that the acceleration would always be increased just because $x-m>s\land v<4$ holds, unlike (3.5), which also checks the additional condition S. Likewise, (3.3) certainly is no faithful model of (3.5). But it looks simpler.

How can we describe a model that is simpler than (3.5) 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. All acceleration should certainly only happen when certain safety-critical conditions are met. But the model should make less commitment than (3.3) about the precise circumstances under which braking is chosen. After all, braking may sometimes just be the right thing to do, for example when arriving at the goal. So we want a model that allows braking under more circumstances than (3.3) without having to model precisely under which circumstances that is. If a system with more behavior is safe, then the actual implementation will be safe as well, because it will only ever exercise some of the verified behavior [10]. The extra behavior in the system might, in fact, occur 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 10 (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 choice operator $\cup$:

$$\begin{aligned}&(a := a+1 \cup a := -b);\\ &\{x' = v, v' = a\}\end{aligned} \tag{3.6}$$

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 11 (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. 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 and a safe system design needs to be prepared to handle either outcome.*

The HP (3.6) is a *faithful abstraction* [10] of (3.5), because every way how (3.5) can run can be mimicked by (3.6) so that the outcome of (3.6) corresponds to that of (3.5). Whenever (3.5) runs $a := a+1$, which happens exactly if $x - m > s \land v < 4 \land S$ is *true*, (3.6) only needs to choose to run the left choice $a := a+1$. Whenever (3.5) runs $a := -b$, which happens exactly if $x - m > s \land v < 4 \land S$ is *false*, (3.6) needs to choose to run the right choice $a := -b$. So all runs of (3.5) are possible runs of (3.6). Furthermore, (3.6) is much simpler than (3.5), because it contains less detail. It does not mention the complicated extra condition S. Yet, (3.6) is a little too permissive, because it suddenly allows the controller to choose $a := a+1$ even when it is already too fast or even at close distance to the obstacle. That way, even if (3.5) was a safe controller, (3.6) is still unsafe, and, thus, not a very suitable abstraction.

3.2.5 Tests in Hybrid Programs

In order to build a faithful yet not overly permissive abstraction of (3.5), we need to restrict the permitted choices in (3.6) so that there is enough flexibility, but only so much that the acceleration choice $a := a+1$ can only be chosen when it is safe to do so. 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.

Of course, it can be difficult to figure out which control choice is safe under what circumstances, and the answer also depends on whether the safety goal is to limit speed or to remain at a safe distance from other obstacles. For the model in this chapter, we simply pretend $v < 4$ would be the appropriate safety condition and revisit the question of how to design and explain such conditions in later chapters.

The test statement can be used to change (3.6) so that it allows acceleration only when $v < 4$, while braking is still allowed always:

$$\begin{aligned}&\big((?v < 4; a := a+1) \cup a := -b\big);\\&\{x' = v, v' = a\}\end{aligned} \tag{3.7}$$

The first statement of (3.7) is a choice ($\cup$) between $\big(?v < 4; a := a+1\big)$ and $a := -b$. All choices in hybrid programs are nondeterministic, so any outcome is always possible. In (3.7), 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 $?v < 4$, which the system run has to pass in order to be able to continue. In particular, if $v < 4$ is indeed *true* in the current state, then the system passes that test $?v < 4$ and the execution proceeds to after the sequential composition (;) to run $a := a+1$. If $v < 4$ is *false* in the current state, however, the system fails the test $?v < 4$ 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 12 (Discarding failed runs) *System runs that fail tests $?Q$ are discarded and not considered any further, because a failed run did not play by the rules of the system. It is as if those failed system execution attempts had never happened. Even if one execution attempt fails, other runs may still be successful. Operationally, you can imagine finding them by backtracking through all the possible choices in the system run and taking alternative choices instead.*

In principle, there are always two choices when running (3.7). Yet, which ones actually run successfully depends on the current state. If the current state is at a far distance from the obstacle (so the test $?v < 4$ will succeed), then both options of accelerating and braking are possible and can execute successfully. Otherwise, only the braking choice executes, because trying the left choice will fail its test $?v < 4$ and be discarded. Both choices formally exist but only one will succeed in that case.

Note 13 (Successful runs) *Notice that only successfully executed runs of HPs will be considered, all others discarded because they did not play by the rules. For example,* $?v < 4; a := a + 1$ *can only run in states where* $v < 4$*, otherwise there are no runs of this HP. Failed runs are discarded entirely, so the HP* $a := a + 1; ?v < 4$ *can also only run in states where* $v < 4$*, in particular, acceleration* a *will never successfully be increased by this HP. Operationally, you can imagine running the HP step by step and rolling all its changes back if any test ever fails. Contrast this with the HP* $a := a + 1; ?a < 6$*, which increments the acceleration and subsequently tests whether* a *is less than 6. This HP can only run successfully from initial states with acceleration at most 5, because their value at the test after the increment will then pass the test* $?a < 6$*.*

Comparing (3.7) with (3.5), we see that (3.7) is a faithful abstraction of the more complicated (3.5), because all runs of (3.5) can be mimicked by (3.7). Yet, unlike the intermediate guess (3.6), the improved HP (3.7) still retains the critical information that acceleration is only allowed by (3.5) when $v < 4$. Unlike (3.5), (3.7) does not restrict the cases where acceleration can be chosen to those that also satisfy $v < 4 \wedge S$. Hence, (3.7) is more permissive than (3.5). But (3.7) is also simpler and only contains crucial information about the controller. Hence, (3.7) is a more abstract faithful model of (3.5) that retains just the relevant detail. Studying the abstract (3.7) instead of the more concrete (3.5) 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 (3.7), which allows lots of behavior, will imply safety of the special concrete case (3.5) but also implies safety of other implementations of (3.7). For example, replacing S by a different condition in (3.5) still gives a special case of (3.7). So if all behavior of (3.7) is safe, all behavior of that different replacement will already be safe. With a single verification result about a more general, more abstract system, we can verify a whole class of systems rather than just one particular system. This important phenomenon [10] 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 chapter. For now, suffice it to say that (3.7) has the relevant level of abstraction for our purposes.

Note 14 (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 simplification 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 crucial, 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 or pedestrians 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 parentheses out without changing (3.7):

$$\begin{aligned}&\big(?v<4;\, a:=a+1 \cup a:=-b\big);\\ &\{x'=v, v'=a\}\end{aligned} \tag{3.7*}$$

3.2.6 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 at some point. Say, one of those controllers, e.g. (3.7), inspects the state and finds it still okay to accelerate. If it chooses $a:=a+1$ and then lets physics move with 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 (3.7) has no way to change its mind, because it has no more choices and cannot exercise any control anymore.

If the controller of (3.7) is supposed to be able to make a second control choice later after physics has followed the differential equation for a while, then (3.7) can simply be sequentially composed with itself:

$$\begin{aligned}&\big((?v<4;\, a:=a+1) \cup a:=-b\big);\\ &\{x'=v, v'=a\};\\ &\big((?v<4;\, a:=a+1) \cup a:=-b\big);\\ &\{x'=v, v'=a\}\end{aligned} \tag{3.8}$$

In (3.8), the cyber controller can first choose to accelerate or brake (depending on whether $v<4$ is true in the present state), 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 $v<4$ is true 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:

$$\begin{aligned}
&\big((?v < 4;\, a := a+1) \cup a := -b\big);\\
&\{x' = v, v' = a\};\\
&\big((?v < 4;\, a := a+1) \cup a := -b\big);\\
&\{x' = v, v' = a\};\\
&\big((?v < 4;\, a := a+1) \cup a := -b\big);\\
&\{x' = v, v' = a\}
\end{aligned} \tag{3.9}$$

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 15 (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.*

The programmatic way of summarizing (3.7), (3.8), (3.9) and the infinitely many more n-fold replications of (3.7) for any $n \in \mathbb{N}$, is by using a repetition operator:

$$\Big(\big((?v < 4;\, a := a+1) \cup a := -b\big);\\ \{x' = v, v' = a\}\Big)^* \tag{3.10}$$

This HP can repeat (3.7) any number of times (0,1,2,3,4,. . .). Of course, it would not be very meaningful to repeat a loop half a time or minus 5 times, so the repetition count $n \in \mathbb{N}$ still has to be some natural number.

But how often does a nondeterministic repetition like (3.10) repeat then? That choice is again nondeterministic.

Note 16 (Nondeterministic ***)** *Repetition (***) is* nondeterministic. *That is, program* α^* *can repeat* α *any number (*$n \in \mathbb{N}$*) of times. 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 (3.10) 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 (3.8), how long does the first $x'' = a$ take before the controller has its second control choice? How long did the continuous evolution take in (3.7) even?

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 according to Chap. 2), it is still a matter of choice how long to follow that solution. The choice is, as always in hybrid programs, nondeterministic.

Note 17 (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 ($0 \leq r \in \mathbb{R}$) within the interval of existence of the solution. 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).*

3.3 Hybrid Programs

Based on the above gradual motivation, this section formally defines the programming language of hybrid programs [15, 17], in which all of the operators motivated above are allowed.

3.3.1 Syntax of Hybrid Programs

Formal grammars have worked well to define the syntax of terms e and first-order logic formulas Q in Chap. 2, which is why we continue to use a grammar to define the syntax of hybrid programs.

Definition 3.1 (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 in x, 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 *assignment* $x := e$ instantaneously changes the value of variable x to the value of e with a discrete state change. The *differential equation* $x' = f(x) \,\&\, Q$ follows a continuous evolution along the differential equation $x' = f(x)$ for any amount of time but restricted to the domain of evolution Q, where x' denotes the time-derivative of x. It goes without saying that $x' = f(x) \,\&\, Q$ is an explicit differential equation, so no derivatives occur in $f(x)$ or Q. Recall that a differential equation $x' = f(x)$ without an evolution domain constraint is short for $x' = f(x) \,\&\, \mathit{true}$, since that imposes no restriction on the duration of the continuous evolution. 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 an *abort* statement would, 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, discarded, 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 successfully). 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 (≥ 0).

Expedition 3.1 (Operator precedence for hybrid programs)

In practice, it is useful to save parentheses by agreeing on notational *operator precedences*. Unary operators (including repetition *) bind stronger than binary operators and ; binds stronger than $\cup$, so $\alpha;\beta \cup \gamma \equiv (\alpha;\beta) \cup \gamma$ and $\alpha \cup \beta;\gamma \equiv \alpha \cup (\beta;\gamma)$. Especially, $\alpha;\beta^* \equiv \alpha;(\beta^*)$.

3.3.2 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 [4]. 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 for programming languages due to Scott and Stratchey [22].

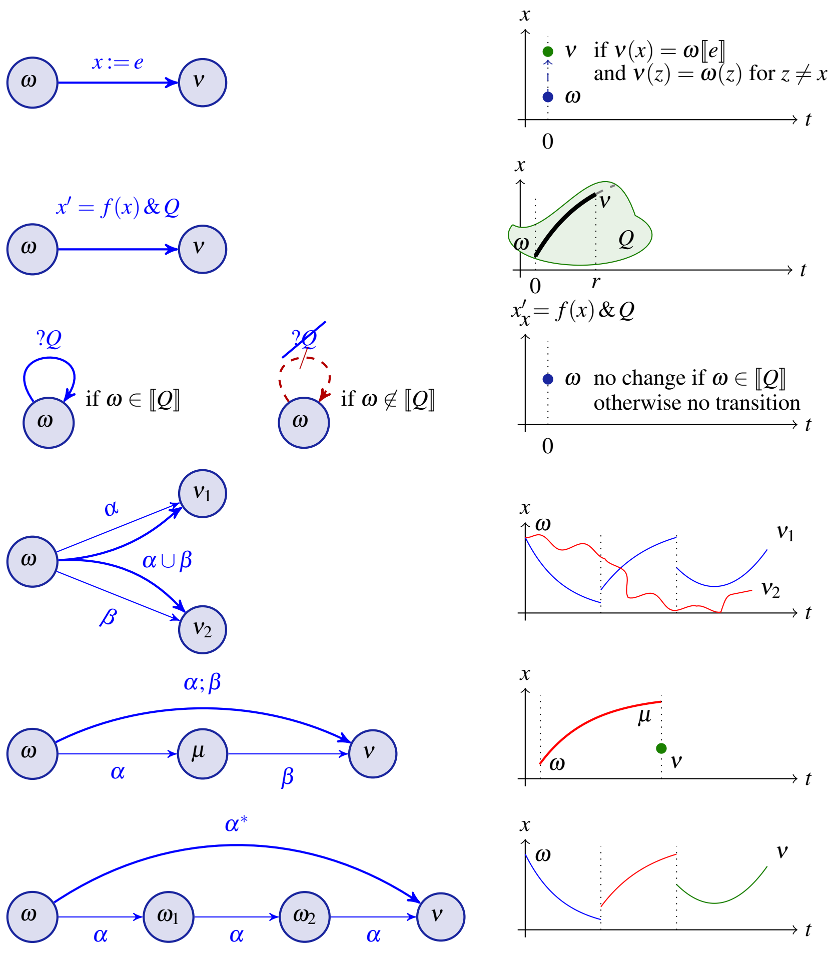
There is more than one way to define the meaning of a program, including defining a denotational semantics [21], an operational semantics [21], a structural operational semantics [19], an axiomatic semantics [6, 20]. For our purposes, what is most relevant is how a hybrid program changes the state of the system. Consequently, the semantics of hybrid programs 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 [8] but can be ignored for most purposes in this book.

Recall that a *state* $\omega : \mathscr{V} \to \mathbb{R}$ is a mapping from variables to $\mathbb{R}$. The set of states is denoted $\mathscr{S}$. The meaning of an HP α is given by a reachability relation $[\![\alpha]\!] \subseteq \mathscr{S} \times \mathscr{S}$ on states. So $(\omega, \nu) \in [\![\alpha]\!]$ 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 states 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 [14–16]. Recall from Chap. 2 that the value of term e in ω is denoted by $\omega[\![e]\!]$ and that $\mathscr{S}$ denotes the set of all states. Further, $\omega \in [\![Q]\!]$ denotes that first-order formula Q is true in state ω. The semantics of an HP α is then defined by its reachability relation $[\![\alpha]\!] \subseteq \mathscr{S} \times \mathscr{S}$. The notation α^* for loops comes from the notation ρ^* for the reflexive, transitive closure of a relation ρ. Graphical illustrations of the transition semantics of hybrid programs defined below and possible example dynamics are depicted in Fig. 3.4. The left of Fig. 3.4 illustrates the generic shape of the transition structure $[\![\alpha]\!]$ for transitions along various cases of hybrid programs α from state ω to state ν. The right of Fig. 3.4 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.

Definition 3.2 (Transition semantics of HPs). Each HP α is interpreted semantically as a binary reachability relation $[\![\alpha]\!] \subseteq \mathscr{S} \times \mathscr{S}$ over states, defined inductively by

1. $[\![x := e]\!] = \{(\omega, \nu) \ : \ \nu = \omega \text{ except that } \nu[\![x]\!] = \omega[\![e]\!]\}$
   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 ω.
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 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. $[\![x' = f(x) \,\&\, Q]\!] = \{(\omega, \nu) : \varphi(0) = \omega \text{ except at } x' \text{ and } \varphi(r) = \nu \text{ for a solution } \varphi : [0, r] \to \mathscr{S} \text{ of any duration } r \text{ satisfying } \varphi \models x' = f(x) \land Q\}$
   That is, the final state $\varphi(r)$ is connected to the initial state $\varphi(0)$ by a



$x := e$
ω
ν
ν if $\nu(x) = \omega[\![e]\!]$ and $\nu(z) = \omega(z)$ for $z \neq x$
$x' = f(x) \,\&\, Q$
Q
r
$?Q$
if $\omega \in [\![Q]\!]$
if $\omega \notin [\![Q]\!]$
ω no change if $\omega \in [\![Q]\!]$ otherwise no transition
α
β
$\alpha \cup \beta$
ν_1
ν_2
$\alpha;\beta$
μ
α^*
ω_1
ω_2
x
t
0

Fig. 3.4 Transition semantics **(left)** and example dynamics **(right)** of hybrid programs

continuous function of some duration $r \geq 0$ that solves the differential equation and satisfies Q at all times; see Definition 3.3.

4. $[\![\alpha \cup \beta]\!] = [\![\alpha]\!] \cup [\![\beta]\!]$
   That is, $\alpha \cup \beta$ can exactly 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. $[\![\alpha^*]\!] = [\![\alpha]\!]^* = \bigcup_{n\in\mathbb{N}} [\![\alpha^n]\!]$ 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 $\alpha^n \equiv \underbrace{\alpha;\alpha;\alpha;\ldots;\alpha}_{n \text{ times}}$ 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, this definition uses simplifying abbreviations for differential equations. Chapter 2 provides full detail, including the definition for differential equation systems. The semantics of loops can also be rephrased equivalently as:

$$[\![\alpha^*]\!] = \bigcup_{n\in\mathbb{N}} \{(\omega_0,\omega_n) : \omega_0,\ldots,\omega_n \text{ are states such that } (\omega_i,\omega_{i+1}) \in [\![\alpha]\!] \text{ for all } i < n\}$$

For later reference, we repeat the definition of the semantics of differential equations separately:

Definition 3.3 (Transition semantics of ODEs).

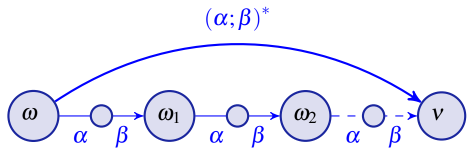
$$[\![x' = f(x)\,\&\,Q]\!] = \big\{(\omega,\nu) : \varphi(0) = \omega \text{ except at } x' \text{ and } \varphi(r) = \nu \text{ for a solution } \varphi : [0,r] \to \mathscr{S} \text{ of any duration } r \text{ satisfying } \varphi \models x' = f(x) \land Q\big\}$$

where $\varphi \models x' = f(x) \land Q$, iff for all times $0 \le z \le r$: $\varphi(z) \in [\![x' = f(x) \land Q]\!]$ with $\varphi(z)(x') \overset{\text{def}}{=} \frac{\mathrm{d}\varphi(t)(x)}{\mathrm{d}t}(z)$ and $\varphi(z) = \varphi(0)$ except at x, x'.

The condition that $\varphi(0) = \omega$ except at x' is explicit about the fact that the initial state ω and the first state $\varphi(0)$ of the continuous evolution have to be identical (except for the value of x', which Definition 3.3 only provides a value for along φ). Part I of this book does not track the values of x' except during continuous evolutions. But that will change in Part II, for which Definition 3.3 is already prepared appropriately.

By plugging one transition structure pattern into another, Fig. 3.4 also illustrates the generic shape of transition structures for more complex HPs, see Fig. 3.5 for an illustration of $(\alpha;\beta)^*$.

Fig. 3.5 Nested transition semantics pattern for $(\alpha;\beta)^*$



Observe that $?Q$ cannot run from an initial state ω with $\omega \notin [\![Q]\!]$, especially $[\![?false]\!] = \emptyset$. Likewise, $x' = f(x)\,\&\,Q$ cannot run from an initial state ω with $\omega \notin [\![Q]\!]$, because no solution of any duration, not even duration 0, starting in ω

will always stay in the evolution domain Q if it already starts outside Q. A nondeterministic choice $\alpha \cup \beta$ cannot run from an initial state from which neither α nor β can run. Similarly, $\alpha;\beta$ cannot run from an initial state form which α cannot run, nor from an initial state from which all final states after α make it impossible for β to run. Assignments and repetitions can always run, e.g., by repeating 0 times.

When α denotes the HP in (3.8) on p. 69, 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 is that, 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 (3.8) could have been one second or two or 424 or half a second or zero or π or any other nonnegative real number. This would have been very different for an HP whose differential equation has an evolution domain constraint, because that limits how long a continuous evolution can take. The exact duration is still nondeterministic, but it cannot ever evolve outside its evolution domain.

3.4 Hybrid Program Design

This section discusses some early lessons on good and bad modeling choices in hybrid systems. As our understanding of the subject matter advances throughout this textbook, we will find additional insights into tradeoffs and caveats. The aspects that can easily be understood on a pure modeling level will be discussed now.

3.4.1 A Matter of Choice

If we change the HP from (3.8) and consider the following modification instead:

$$\begin{aligned}&?v < 4;\, a := a+1;\\ &\{x' = v, v' = a\};\\ &?v < 4;\, a := a+1;\\ &\{x' = v, v' = a\}\end{aligned} \tag{3.11}$$

Then some behavior that was still possible in (3.8) is no longer possible for (3.11). Let β denote the HP in (3.11), 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). Note that the duration of the first differential equation in (3.11) 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 $?v < 4$, which, according to the semantics of tests, will fail and be discarded.

That is what makes (3.11) a bad model, because it truncates and discards behavior that the real system would still possess. Even if the controller in the third line of (3.11) is not prepared to handle the situation where the test $?v < 4$ fails, it might fail in reality. In that case, the controller in (3.11) simply ran out of choices. A more realistic and permissive controller, thus, also handles the case if that test fails, at which point we are back at (3.8).

Note 18 (Controllers cannot discard cases) *While subsequent chapters discuss cases where hybrid programs use tests $?Q$ in crucial ways to discard non-permitted behaviors, great care needs to be exercised that controllers also handle the remaining cases. A bad controller*

$$?x > s;\ \alpha$$

only handles the case where $x > s$ and ignores all other circumstances, which renders the controller incapable of reacting and, thus, unsafe when $x \leq s$. A better controller design always considers the case when a condition is not satisfied and handles it appropriately as well:

$$(?x > s; \alpha) \cup (?x \leq s; \dots)$$

Liveness proofs can tell both cases of controllers apart, but appropriate design principles of being prepared for both outcomes of each test go a long way in improving the controllers.

Similarly bad controller designs result from careless evolution domains:

$$a := -b;\ \{x' = v, v' = a \,\&\, v > 4\}$$

The differential equations in this controller silently assume the velocity would always stay above 4, which is clearly not always the case when braking. Accidental divisions by zero are another source of trouble in CPS controllers.

3.4.2 *Patternology?*

3.5 Summary

This chapter introduced hybrid programs as a model for cyber-physical systems, summarized in Table 3.1. 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.

Table 3.1 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

3.1. The semantics of an HP α is its reachability relation $[\![\alpha]\!]$. For example,

$$[\![x:=2\cdot x;x:=x+1]\!]=\{(\omega,\nu)\ :\ \nu(x)=2\cdot\omega(x)+1 \text{ and } \nu(z)=\omega(z) \text{ for all } z\neq x\}$$

Describe the reachability relation of the following HPs in similarly explicit ways:

1. $x:=x+1;x:=2\cdot x$
2. $x:=1\cup x:=-1$
3. $x:=1\cup ?(x\leq 0)$
4. $x:=1;\ ?(x\leq 0)$
5. $?(x\leq 0)$
6. $x:=1\cup x'=1$
7. $x:=1;\ x'=1$
8. $x:=1;\ \{x'=1\,\&\,x\leq 1\}$
9. $x:=1;\ \{x'=1\,\&\,x\leq 0\}$
10. $v:=1;\ x'=v$
11. $v:=1;\ \{x'=v\}^*$
12. $x'=v,v'=a\,\&\,x\geq 0$

3.2. The semantics of hybrid programs (Definition 3.2) 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?

3.3 (If-then-else). Sect. 3.2.3 considered if-then-else statements for hybrid programs. But they no longer showed up in the grammar of hybrid programs. Is this a mistake? Can you define $\mathsf{if}(P)\,\alpha\,\mathsf{else}\,\beta$ from the operators that HPs provide?

3.4 (If-then-else). Suppose we add the if-then-else-statement $\mathsf{if}(P)\,\alpha\,\mathsf{else}\,\beta$ to the syntax of HPs. Define a semantics $[\![\mathsf{if}(P)\,\alpha\,\mathsf{else}\,\beta]\!]$ for if-then-else statements.

3.5 (Switch-case). Define a switch statement that runs the statement α_i if formula P_i is true, and chooses nondeterministically if multiple conditions are true:

```
switch (
    case $P_1$ : $\alpha_1$
    case $P_2$ : $\alpha_2$
        ⋮
    case $P_1$ : $\alpha_1$
)
```

What would need to be changed to make sure only the statement α_i with the first true condition P_i executes?

3.6 (While). Suppose we add the while loop $\mathsf{while}(P)\,\alpha$ to the syntax of HPs. As usual, $\mathsf{while}(P)\,\alpha$ is supposed to run α if P holds, and, whenever α finished, repeat again if P holds. Define a semantics $[\![\mathsf{while}(P)\,\alpha]\!]$ for while loops. Can you define a program that is equivalent to $\mathsf{while}(P)\,\alpha$ from the original syntax of HPs?

3.7. 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? And what would you need to add to your programming language to cover all of hybrid systems? How would you best do that?

3.8. Can you find a discrete controller *ctrl* and a continuous program *plant* such that the following two hybrid programs have very different behavior?

$$(ctrl;plant)^* \quad \text{versus} \quad (ctrl \cup plant)^*$$

3.9 (Set-valued semantics). The semantics of hybrid programs (Definition 3.2) is defined as a transition relation $[\![\alpha]\!] \subseteq \mathscr{S} \times \mathscr{S}$ on states. Define an equivalent semantics using functions $R(\alpha) : \mathscr{S} \to 2^{\mathscr{S}}$ from the initial state to the set of all final states, where $2^{\mathscr{S}}$ denotes the powerset of $\mathscr{S}$, i.e. the set of all subsets of $\mathscr{S}$. Define this set-valued semantics $R(\alpha)$ without referring to the transition relation semantics $[\![\alpha]\!]$ and prove that it is equivalent, i.e.

$$\nu \in R(\alpha)(\omega) \quad \text{iff} \quad (\omega,\nu) \in [\![\alpha]\!]$$

Likewise, define an equivalent semantics based on functions $\varsigma(\alpha) : 2^{\mathscr{S}} \to 2^{\mathscr{S}}$ from the set of possible final states to the set of initial states that can end in the given set of final states. Prove that it is equivalent, i.e. for all sets of states $X \subseteq \mathscr{S}$:

$$\omega \in \varsigma(\alpha)(X) \quad \text{iff} \quad \text{there is a state } \nu \in X \text{ such that } (\omega,\nu) \in [\![\alpha]\!]$$

3.10 (Switched systems). Hybrid programs come in different classes; see Table 3.2. According to Chap. 2, for example, a continuous program is an HP that only consists of one continuous evolution of the form $x' = f(x) \,\&\, Q$. A discrete system corresponds to an HP that has no differential equations. A switched continuous system corresponds to an HP that has no assignments, because it does not have any instant changes of state variables but merely switches mode (possibly after some tests) from one continuous mode into another.

Table 3.2 Classification of hybrid programs and correspondence to dynamical systems

HP class	Dynamical systems class
only ODE	continuous dynamical systems
no ODE	discrete dynamical systems
no assignment	switched continuous dynamical systems
general HP	hybrid dynamical systems

Consider an HP in which the variables are partitioned into state variables (x, v) sensor variables (m) and controller variables (a):

$$\Big(\big((?v < 4;\, a := A) \cup a := -b\big);\\ \{x' = v, v' = a\}\Big)^*$$

Transform this HP into a switched program that has the same behavior on the observable state and sensor variables but is a switched system, so does not contain any assignments. The behavior of controller variables is considered irrelevant as long as the behavior of the other state variables x, v is unchanged.

3.11 (Nondeterministic assignments). Suppose we add a new statement $x := *$ for nondeterministic assignment to the syntax of HPs. The nondeterministic assignment $x := *$ assigns an arbitrary real number to the variable x. Define a semantics $[\![x := *]\!]$ for the $x := *$ statement.

3.12 (Program interpreter).** In a programming language of your choosing, fix a recursive data structure for hybrid programs from Definition 3.1 and fix some finite representation for states where all variables have rational values instead of reals. Write a program interpreter as a computer program, which, given an initial state ω and a program α, successively enumerates possible final states ν that can be reached by α from ω, that is $(\omega, \nu) \in [\![\alpha]\!]$ by implementing Definition 3.2. Resolve nondeterministic choices in the transition either by user input or by randomization. What makes the case of differential equation case particularly challenging?

References

1. Alur, R., Courcoubetis, C., Henzinger, T. A. & Ho, P.-H. *Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems.* in *Hybrid Systems* (eds Grossman, R. L., Nerode, A., Ravn, A. P. & Rischel, H.) **736** (Springer, 1992), 209–229.
2. Church, A. A Note on the Entscheidungsproblem. *J. Symb. Log.* **1,** 40–41 (1936).
3. David, R. & Alla, H. On Hybrid Petri Nets. *Discrete Event Dynamic Systems* **11,** 9–40 (2001).
4. Frege, G. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens* (Verlag von Louis Nebert, 1879).
5. Harper, R. *Practical Foundations for Programming Languages* 2nd ed. doi:`10.1017/CBO9781316576892` (Cambridge Univ. Press, 2016).
6. Hoare, C. A. R. An Axiomatic Basis for Computer Programming. *Commun. ACM* **12,** 576–580 (1969).
7. Hopcroft, J. E., Motwani, R. & Ullmann, J. D. *Introduction to Automata Theory, Languages, and Computation* 3rd ed. (2006).
8. Jeannin, J. & Platzer, A. *dTL^2: Differential Temporal Dynamic Logic with Nested Temporalities for Hybrid Systems* in *IJCAR* (eds Demri, S., Kapur, D. & Weidenbach, C.) **8562** (Springer, 2014), 292–306. doi:`10.1007/978-3-319-08587-6_22`.
9. *Proceedings of the 27th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25–28, 2012* (IEEE, 2012).
10. Loos, S. M. & Platzer, A. *Differential Refinement Logic* in *LICS* (eds Grohe, M., Koskinen, E. & Shankar, N.) (ACM, 2016), 505–514. doi:`10.1145/2933575.2934555`.
11. Nerode, A. & Kohn, W. *Models for Hybrid Systems: Automata, Topologies, Controllability, Observability* in *Hybrid Systems* (Springer-Verlag, London, UK, UK, 1993), 317–356.
12. Olderog, E.-R. *Nets, Terms and Formulas: Three Views of Concurrent Processes and Their Relationship* 267 (Cambridge University Press, 1991).
13. Platzer, A. *Differential Dynamic Logic for Verifying Parametric Hybrid Systems.* in *TABLEAUX* (ed Olivetti, N.) **4548** (Springer, 2007), 216–232. doi:`10.1007/978-3-540-73099-6_17`.
14. Platzer, A. Differential Dynamic Logic for Hybrid Systems. *J. Autom. Reas.* **41,** 143–189 (2008).
15. Platzer, A. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics* doi:`10.1007/978-3-642-14509-4` (Springer, Heidelberg, 2010).
16. Platzer, A. *Logics of Dynamical Systems* in *LICS* (IEEE, 2012), 13–24. doi:`10.1109/LICS.2012.13`.
17. Platzer, A. *The Complete Proof Theory of Hybrid Systems* in *LICS* (IEEE, 2012), 541–550. doi:`10.1109/LICS.2012.64`.

18. Platzer, A. A Complete Uniform Substitution Calculus for Differential Dynamic Logic. *J. Autom. Reas.* doi:10.1007/s10817-016-9385-1 (2016).
19. Plotkin, G. D. *A structural approach to operational semantics* tech. rep. DAIMI FN-19 (Aarhus University, Denmark, 1981).
20. Pratt, V. R. *Semantical Considerations on Floyd-Hoare Logic* in *17th Annual Symposium on Foundations of Computer Science, 25-27 October 1976, Houston, Texas, USA* (IEEE, 1976), 109–121.
21. Scott, D. S. *Outline of a Mathematical Theory of Computation* Technical Monograph PRG–2 (Oxford University Computing Laboratory, Oxford, England, Nov. 1970).
22. Scott, D. & Strachey, C. *Toward a mathematical semantics for computer languages?* tech. rep. PRG-6 (Oxford Programming Research Group, 1971).
23. Turing, A. M. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society 2* **42,** 230–265 (1937).