15-424/15-624/15-824 Recitation 2
Logic and transition relations

1. **Announcements**

   - Theory 1 and Lab 1 are released on the course web page.
   - Course schedule reminder: BetaBot labs are due on **Fridays before recitation** and **you may not use late days for BetaBots!**
   - Theory 0 is now graded. The feedback should be available as an annotated PDF on Autolab. Please ask on Piazza if you can't find it.

2. **Motivation and Learning Objectives**

   Safety properties of Cyber-physical Systems are often stated in terms of *reachability* – "starting from an initial safe state, every state the system can *reach* satisfies a safety property". This informal prose translates into a $\mathsf{d}\mathcal{L}$ formula of the form

   $$\psi \rightarrow [\alpha]\phi$$

   where

   - ψ describes initial state(s) of the system,
   - α is a hybrid program describing the behavior of the system, and
   - ϕ is the safety property.

   In order to give a formal and unambiguous meaning to formulas of this form, we need to describe in a mathematically rigorous way what it means for a state ω to be *reachable* from an initial state ν by a hybrid program α. This recitation presents a **transition relation for hybrid programs** and a diagrammatic representation of the transition relation that serves this purpose.

   By the end of this recitation, you should:

   (a) Know how to draw the transition diagram corresponding to a hybrid program.

   (b) Know how to write down the hybrid program corresponding to a transition diagram.

   (c) Know how to avoid common modeling mistakes that occur when a hybrid program has an empty transition relation.

   (d) Identify sources of non-determinism in hybrid programs.

   (e) Understand the transition relation for hybrid programs.

   We will also cover some basic facts about the KeYmaera X system that you need to understand for Lab 1.

3. **Reminder: satisfiability and validity**

Let ϕ be a formula. Recall that ϕ is

- *Satisfiable* if there is a state ν such that $\nu \models \phi$.
- *Valid* if that for all ν, $\nu \models \phi$
- *Falsifiable*, or *not valid*, if there is a ν such that $\nu \not\models \phi$
- *Unsatisfiable* if there is no ν such that $\nu \models \phi$ [1]

Notice how these notions critically depend on ν, which states the *initial* values for variables. So, if we look at simple formula $x < y$, then to figure out whether it is valid, satisfiable or unsatisfiable we simply try to find a state – an assignment of variables – that satisfies or falsifies it.

In this case, $x > y$ is satisfiable but not valid. That's because if we consider the state $\nu = \{x \mapsto 2, y \mapsto 1\}$, then $\nu \models x > y$, but we can also find $\omega = \{x \mapsto 1, y \mapsto 2\}$ where $\omega \not\models x > y$.

How do quantifiers affect satisfiability and validity? Let's change our formula to $\forall x.x > y$. The semantics state that

$$\nu \models \forall x.x > y \text{ iff } \nu[x \mapsto d] \models x > y \textit{ for all } d \in \mathbb{R}$$

Really, the quantifier is overwriting whatever value ν originally assigned to x. The same happens for the existential quantifier!

$$\nu \models \exists x.x > y \text{ iff } \nu[x \mapsto d] \models x > y \textit{ for some } d \in \mathbb{R}$$

Is this existential formula $\exists x.x > y$ satisfiable? It is, because we can find an assignment of variables, like $\nu = \{x \mapsto 1, y \mapsto 2\}$ that satisfies it. Even though $\nu \not\models x > y$, the real question, because of the quantifier, is whether we can overwrite x with a new value that is larger than y. The answer is yes, since we can choose x to be 3.

Now imagine that there aren't any free variables, i.e. that all variables are quantified, like for example:

$$\nu \models \forall x.\forall y.\phi(x, y)$$

Above, $\phi(x, y)$ is any formula that depends on x and y. This formula can never be satisfiable without being valid. That's because it doesn't really matter what the initial values ν assigns to variables, since they will always be overwritten by the quantifiers.

So formulas without free variables are always either valid or unsatisfiable.

[1] Perhaps it is more appropriate to call them *Rolling Stones*, because they can't get no satisfaction.

Notation for State Updates

A fundamental operation in the semantics of differential dynamic logic is updating the value of a single variable in a state ν. In this course we'll use both $\nu[x \mapsto d]$ and ν_x^d to denote this operation. Both of these notations means the same thing – "ν except that the value of x is d instead of its previous value". For example, if $\nu = \{x \mapsto 2, y \mapsto 1, z \mapsto 22.4523\}$ then $\nu_x^d = \nu[x \mapsto d] = \{x \mapsto d, y \mapsto 1, z \mapsto 22.4523\}$.

4. **Transition Diagrams**

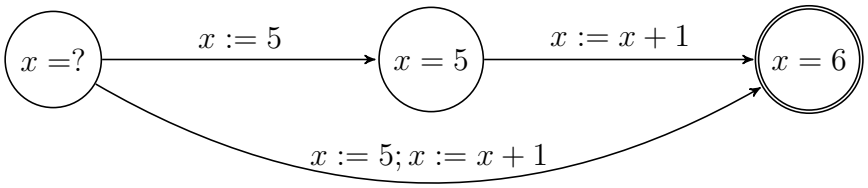
Hybrid programs are given meaning by defining a *transition relation*, which is a function that describes how hybrid programs change state.

Transition diagrams are a graph-like representation of transition relations. Two nodes ν and ω of a transition diagram are connected by an edge labeled with α whenever the state ω is reachable by executing the program α in state ν.

Assignment and Sequential Composition For example, consider the program:

$$x := 5; x := x + 1$$

The transition diagram corresponding to this program is



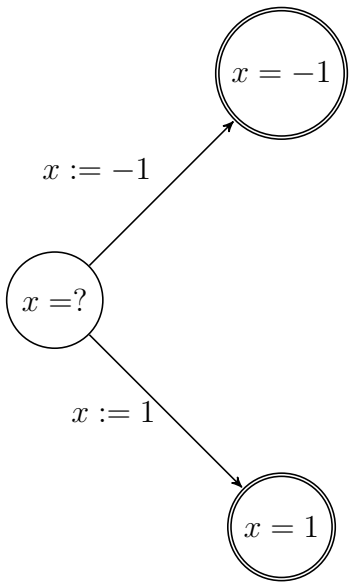
No matter what value x has in state ν ($\nu = \{x =?\}$), executing $x := 5$ in state ν results in a state where $x = 5$ ($\omega = \{x = 5\}$). Sequential composition of two hybrid programs $(\alpha; \beta)$ is denoted in a transition diagram by first following all edges annotated with an α and then all the edges annotated with β. The double lines around the final state $(x = 6)$ indicate that this state is **final**; i.e., the state is reached after executing the entire hybrid program. Note that the $x = 5$ state is intermediate, not final, because $x := x + 1$ has not yet executed.

A Syntax Note: In the above diagram, there are a few different kinds of "equality" symbol. := is the *assignment operator*, which only appears in programs. It has the effect of changing the value of some variable, e.g. x. On the other hand, when we write $x = y$ (with a colon), this an *equality formula*, which asks a questions: Is x equal to y right now or not? It never changes anything. But sometimes = appears in programs too! If we write $x' = \theta \& H$ or even just $x' = \theta$, this is a hybrid program describing a differential equation.

Choice Most hybrid programs have some amount of *non-determinism* – i.e., situations where *multiple* states are reachable by the same hybrid program from a given initial state. Transition diagrams represent non-determinism using branching. For example, consider a simple hybrid program that contains the choice operator:

$$x := -1 \cup x := 1$$

The program will assign to x either the value -1 or the value 1. We give meaning to non-determinism by allowing *multiple* states to be reachable via a hybrid program.



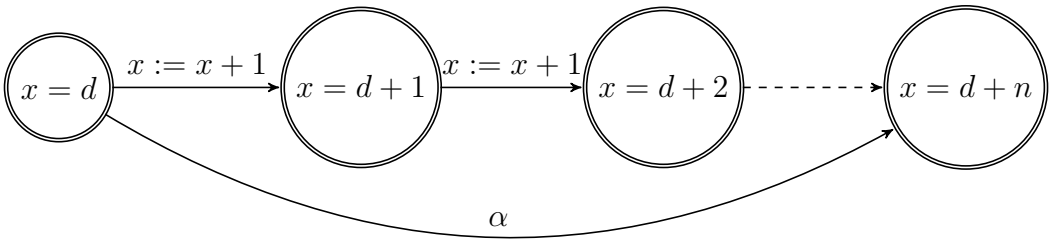
Iteration The choice operator ($\cup$) isn't the only source of non-determinism in hybrid programs. Recall that $\{x := x + 1\}^*$ means "execute $x := x + 1$ zero or more times". So the transition relation for

$$\alpha \equiv \{x := x + 1\}^*$$

should map the state $\{x = d\}$ to:

- The state $\{x = d\}$ (by executing $x := x + 1$ zero times) and
- The state $\{x = d + 1\}$ (by executing $x := x + 1$ one time) and
- The state $\{x = d + 2\}$ (by executing $x := x + 1$ two times) and...
- (Hopefully you see the pattern now!)

The transition diagram corresponding to this program is:



Notice that all of the states are final.

Tests Test operations ($?\varphi$) either halt the program if the formula φ is not true, or else allow the program to continue executing. In particular, the state never changes when a test operation succeeds (or fails)! We can draw this in a picture using a *self-loop* whenever φ evaluates to true:

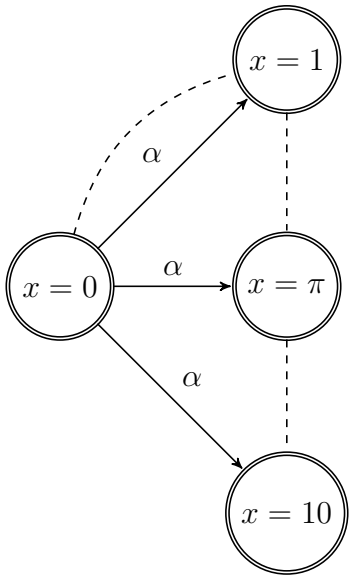
Left: Successful test, **Right:** Failed test

Note what happens when a test fails – the transition relation for α from state ν will contain *no* transitions! We'll return to this in a later section.

Differential Equations Differential equations are a third source of non-determinism in hybrid programs. Consider the program

$$\alpha \equiv x' = 1 \& x \leq 10$$

starting in state $x = 0$:

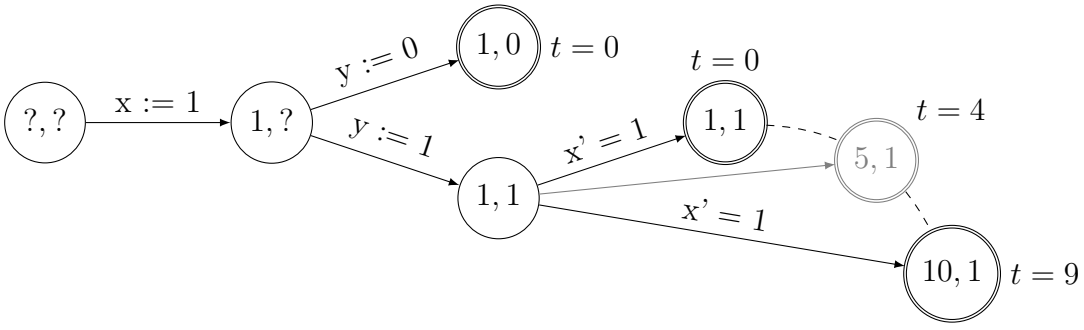


The transition relation for differential programs includes *all* states that are reachable by following the solution of the differential equation while staying within the evolution domain constraint for all time. (There are quite a lot of states that aren't explicitly represented in this diagram!)

Putting it all together Consider the following program:

$$\alpha = x := 1; y := 0 \cup (y := 1; x' = 1 \ \& \ x \leq 10)$$

If you think about starting in a specific state, ν_0, it's not too hard to look at the transition relation as if it was a tree.



In the tree above, each node is a state, and the two numbers inside each node are the possible values for x, y, respectively. Notice how the tree doesn't all have to be the same depth: that's because certain paths along the program are simply shorter!

There are three sources of non-determinism in hybrid programs, and they have different branching factors:

choice ($\alpha \cup \beta$)	Finite (k for some $k \in \mathbb{N}$)
looping (α^*)	Countable ($\lvert\mathbb{N}\rvert$)
differential equations (α^*)	Uncountable ($\lvert\mathbb{R}\rvert$)

The first two are called **structural non-determinism** because they and do not represent any passage in time, they just help us structure a dL model by composing it from smaller pieces. Differential equations, however, are a *continuous* choice that does represent passage in time.

If you were to remove the domain of the differential equation, that transition would be able to evolve for *any* duration. [2]

5. **The Transition Relation** Now we're comfortable with transition diagrams, but I argue we should be a little uncomfortable. These are more rigorous than just talking about hybrid programs, but they are not nearly as precise as math.

   So now we're going to give the formal definition of hybrid programs as a transition relation. This is the denotational semantics mentioned in the previous recitation: We want to know precisely what hybrid programs mean, so we explain them by reducing them to math we[3] already know.

   Specifically, we want to define a function of type

   $$R : HP \to V \to 2^V$$

   where V is the set of all states and 2^V is the *powerset* of V.

   The transition relation maps states to sets of states because of non-determinism – the program $x := 1 \cup x := 2$ possibly transitions to two states from a given state ν, so $R(x := 1 \cup x := 2)(\nu) = \{\{x = 1\}, \{x = 2\}\}$.

   The transition relation for non-differential programs is pretty straight-forward:

   - $R(x := \theta)(\nu) = \{\nu[x \mapsto [\![\theta]\!]_\nu]\}$
   - $R(?H)(\nu) = \{\nu : \nu \models H\}$
   - $R(\alpha \cup \beta)(\nu) = R(\alpha)(\nu) \cup R(\beta)(\nu)$
   - $R(\alpha^*)(\nu) = \bigcup_{n \in \mathbb{N}_{\geq 0}} R(\alpha^n)(\nu)$

   For differential equations, assume we have a solution $\varphi : [0, t] \to S$ to the *initial value problem*, where initial values come from an **initial** state ν. Starting from ν, which states will we be able to reach? The idea here is to follow the solution φ for as long as possible, until we fail to satisfy H. As we evolve, we collect all the states that we've been passing through and add them to the transition relation. Since the we allow the differential equation to stop at any time, then any state we pass through could've been an end state!

[2] Assuming the ODE has a solution for all time t, otherwise it can evolve so long as the solution exists.
[3] For generalized values of "we"

$$R(x' = f(x) \;\&\; H)(\nu) = \{\varphi(t) : \varphi \text{ is a sol. on } [0,t] \;\wedge\; \varphi(0) = \nu \wedge \forall_{0\leq r\leq t}.\; \varphi(r) \models H\}$$

So really, as long as we are along the φ path, we add our current location to the set of end states as long as we've been within the domain H since the start!

6. **Avoid Modeling Mistakes: Understand Tests in Boxes** Now we've talked about what programs mean formally, but if we want to avoid writing bogus models (which we do), we need to investigate what happens when the formal meaning disagrees with the intuition we (might) have.

   Let's look at what happens when we put tests in box. Recall that $[\alpha]\varphi$ informally means φ is true in every possible end state of the program α. Formally, we say $[\alpha]\varphi$ is satisfiable if there exists a state ν such that

   $$\nu \models [\alpha]\varphi$$

   Now that we have a transition semantics, we can be a little less hand-wavy about what exactly $[\alpha]\varphi$ is supposed to mean!

   Let's test this new understanding on a simple-looking formula:

   $$[?(x > 0)]1 = 0$$

   Is this formula satisfiable? Surprisingly, **yes!** Suppose we choose the initial state to be $\{x = -1\}$. Then $?(x > 0)$ has *no* transitions, and $1 = 0$ is true in each of those (zero) states! That is, $[?(x > 0)]1 = 0$ holds vacuously!

   In this case, we could also think of the problem in terms of first-order logic. The formula $[?(x > 0)]1 = 0$ is equivalent to $x > 0 \rightarrow 1 = 0$, which will be true any time $x > 0$ doesn't hold. More generally, the rule for any hybrid program is easy to remember: if $R(\alpha)(\nu) = \{\}$ then $\nu \models [\alpha]\varphi$ must be true — it doesn't actually matter what φ is.

   You can test this intuition by loading the following model into KeYmaera X:
   Note: This was rushed at the end due to projector issues in recitation. Reading through this part of the notes may help clear it up.

```
ProgramVariables.
  R x.
End.
Problem.
  x = -1 ->
  [
    ?x > 0;
  ] (1 = 0)
End.
```

and execute the tactic:

```
master
```

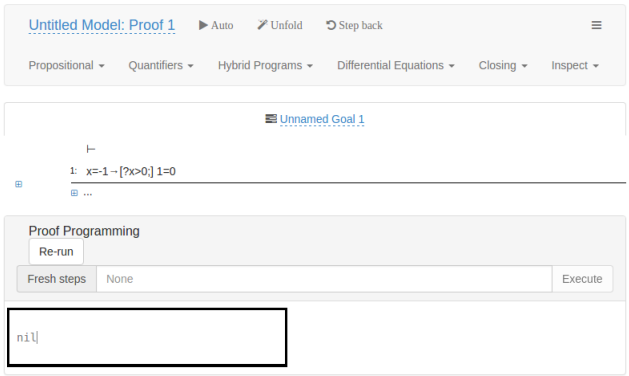
by deleting `nil` in the Proof Programming section, replacing it with `master` and pressing "Execute". This is a good time to explain the "Proof Programming" area in general. So far you've only worked with very simple models that automatically prove by pressing the "auto" button or running the "master" tactic. However, in the next couple of labs you'll have to build very large proofs. For large proofs, you won't want to do the entire proof by clicking (one mistake and you have to re-do 10 minutes of clicking!).

The programs you see/write in the Proof Programming area are called *tactics*, which simply means they're programs that do a proof for us when we run them. These are useful for several reasons:

- They make repetitive proofs a lot easier. If your proof has lots of repetitive cases, you can prove the first case by clicking, then copy-paste the tactic on the rest of the cases. If you're lucky, the same tactic will prove them all. If you're not lucky, you can still often do the proof by making small changes to the previous tactic.
- Tactics are nice way to write down your proof textually. This is how I grade your labs: by reading the tactic that KeYmaera X produced.
- Tactics enable to write fancy programs to prove things automatically by building them up from smaller tactics. For example, ODE solving and even the play button are just tactics. You probably don't want to write an ODE solver on the labs, but you may find some reusable snippets that make your proofs nicer.

So tactics are great, but how do we learn how to write them? We'll get a more thorough introduction in next week's recitation, but you can also learn them by just watching the Proof Programming area when you do steps on the UI. There a lot of different tactics in KeYmaera X, so this can help you learn all their names.

Once you have your tactic (by copy-pasting and/or editing it in your favorite text editor), you can paste it into the Proof Programming box inside the following black box:



When you type or paste into the box, it will automatically detect which parts of your tactic have changed and show the changes in "Fresh Steps".

Clicking "Execute" in the editor will run the new steps of the tactic.

We'll teach you more about tactic scripting in the next recitation. For Lab 1, just remember that master is very useful for simple models!

7. **Debugging the Bouncing Ball** Let's get some practice looking for modeling mistakes in a real model. Let's debug a bouncing ball model similar to the one from lecture:

$$[\{\{x' = v, v' = -g, x \geq 0\}; ?(x = 0); x := -c * v\}^*](x \leq H)$$

There are many flaws in this model, but they have solutions:

- What if the ball was dropped from a height higher than H? (Add precondition: $x = H$ or $x \leq H$)
- What if the ball was thrown up in the air? (Precondition $v \leq 0$)
- What if the ball was thrown hard at the floor? (Precondition $v = 0$)
- What if the ground is weird, e.g. a trampoline? Add precondition: $0 < c \leq 1$.
- What if the ball is buried? ($x \geq 0$)
- What if the ball is being thrown down stairs? (H and 0 better be constant, but it's ok because they are)
- What if gravity goes up? (precondition $g \geq 0$). Remember that g is just a variable, not anything special, so we need to say if we want it to have some special property.

The point: modeling is subtle! Sometimes when you can't prove a formula, it's because it's false![4] Conversely, it's perfectly possible to prove properties about formulas that don't accurately model any sort of real-life scenario. That's why we put so much effort into picking the right model!

8. **Always remember to check for an update by looking at the version info text in the footer of every page on the KeYmaera X Web UI! We'll be updating often.**

Don't forget about Lab 1 and Theory 1!
Office hours are posted on the course website.

9. **More Semantics: Formulas** We talked about semantics of programs, but we haven't had much of a chance to talk about semantics of formulas. This is important for all the same reasons, as is defined similarly, except it's actually easier. The meaning of

[4] If it's false but you can still prove it, you should let us know for extra credit.

a formula is just "what states make me true" so the semantics can be a set of states, written like $[\![\varphi]\!]_\omega$, i.e. $[\![\_]\!] : \text{Formula} \to 2^{\text{State}}$. We give the definition inductively like before:

- $[\![\neg\phi]\!] = [\![\phi]\!]^C$
- $[\![\phi \wedge \psi]\!] = [\![\phi]\!] \cap [\![\psi]\!]$
- $[\![\phi \vee \psi]\!] = [\![\phi]\!] \cup [\![\psi]\!]$
- $[\![\phi \to \psi]\!] = [\![\psi]\!] \backslash [\![\phi]\!]^C$
- $[\![\forall x \phi]\!] = \{\omega . \forall r \in \mathbb{R} . \omega_x^r \in [\![\phi]\!]\}$
- $[\![\exists x \phi]\!] = \{\omega . \exists r \in \mathbb{R} . \omega_x^r \in [\![\phi]\!]\}$
- $[\![[\alpha]\phi]\!] = \{\omega . \forall \nu \in R(\alpha)(\omega) . \nu \in [\![\phi]\!]\}$
- $[\![\langle\alpha\rangle\phi]\!] = \{\omega . \exists \nu \in R(\alpha)(\omega) . \nu \in [\![\phi]\!]\}$

Thinking in terms of sets like this not only gives us a precise description of truth for formulas, it also gives an alternate description of validity, unsatisfiablity and satisfiability.

In terms of sets:

- ϕ is unsatisifiable if $[\![\phi]\!] = \emptyset$
- ϕ is satisfiable if $[\![\phi]\!] \neq \emptyset$
- ϕ is valid if $[\![\phi]\!] = S$ (the set of all possible states)