David Adam Spencer
Date Created: 6/18/2015

LINEAR TEMPORAL LOGIC

SUMMARY

With LTL you will generally see it introduced with this equation.

$$\phi ::= a|True|\neg\phi|\phi_1 \wedge \phi_2|\phi_1 U \phi_2$$

There may be more or less options in the equation depending on what the person is using the logic for. This is a very confusing equation for us control guys because it doesn't look like it means anything, and that's because it really doesn't when it comes to dynamics. This equation is merely for defining the language of LTL and the basic components necessary to use the language. Think of it as saying the logical specifications ϕ are defined by propositions, a, which are basically Boolean arguments (e.g. $x > 100$, "light is red", etc.), defined by Boolean logic $True, False = \neg True$, the "not" operator, $\neg$, the "and" operator, $\wedge$, and the "until" operator, U. Most of these work just as you would intuitively expect. For U see the example specification below for some more description

You will also see the definition of certain syntaxes that use the above definitions to define further definitions (remember, the above equation are the "basic" building blocks). For instance:

Eventually: $\Diamond \phi = TrueU\phi$ The specification is satisfied if there is a future point where ϕ is true.

Always: $\Box\phi = \neg \Diamond \neg\phi$ ϕ holds for the current and every future state, i.e. it is "never eventually not true", no future state makes ϕ false.

There are also others such as the "or" operator, $\vee$, implication, equivalence, and next (their definitions are mostly intuitive and can be easily found online).

Here is an **example**:

Define the LTL specification

$$\phi \vDash \Diamond\, (x > 10) \wedge (\neg(y < 0) U \text{ "system is ready"})$$

What this says is that our system should be such that the system should eventually reach the state where the variable x exceeds 10 AND y does not fall below zero UNTIL the system enters its "ready" state. This "ready" state could be indicated by x, some other variable, or a nonmathematical action such as "operator turns key" that transitions the system from "standby" to "ready" (via a labeled transition system (LTS)). **Note** that once the system enters "ready" y is *free* to drop below zero but is not required to. It is simply now *allowed* to.

A more complex **example** for us control guys:

$$\phi \vDash \Diamond\, A \wedge \Box(\Diamond\, B \wedge \Diamond\, C) \wedge \Box\neg\, D$$

Let $A, B, C,$ and D be regions in the state space. What this specification says is that your system should eventually reach region A at least once AND ALWAYS be going between B and C AND ALWAYS avoids D, which could be an obstacle or a danger region.

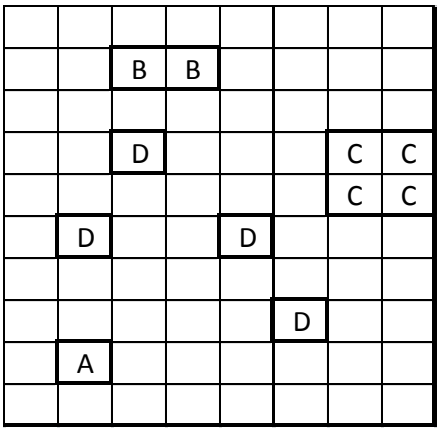
But how can a model checker prove this exists for a system with *continuous* (infinite) states?

David Adam Spencer
Date Created: 6/18/2015

Using Model Checking (Setup)

In order to make the above specification to be any use to us in controls, we need an approach to prove whether or not it is true for our system. The model checker uses a "states and transitions" approach. This is like a labeled transition system or an automaton. For us that requires creating an *abstraction* of the system; that is to say we need to discretize the state space of the system into discrete equivalence regions where certain propositions are held throughout. For instance, we can discretize the system from our above example into subsets of the overall domain such that the proposition "$in\ A$" is its own discrete state (i.e. $A = \{1 < x < 2, 1 < y < 2\}$) and we turn this into a "cell" in the state space.

Perhaps the easiest way we can do such an abstraction is by creating a grid of the state space. The below example is for a straightforward grid of an x-y plane, but this can be extended to use other methods (such as triangulation) in R^n spaces.



By using this cell structure, it is then possible to set up the transitions as moving to an adjacent cell. For instance, if you are in the cell marked by A, you can move up, down, left, or right according to the abstracted model.

You can then set up this abstracted model into the model checker (NuSMV in our case) along with the above specification to test whether the specification is satisfied. Now how can we do that?

Using the Model Checker (Conceptually)

The way the model checker works is proof by contradiction. It will take the model and try to find a path where the given specification is not satisfied. If no path is found, the system satisfies the specification. If we were to use the example specification and the above example with the model checker it would simply return a case where it moves repeatedly between the first and second cells, makes a beeline for a D region, or any other number of ways the specification is shown to be false.

So how can the model checker be useful at all, then? Well, in this case we aren't using the model checker to check if our model satisfies ϕ everywhere, we only want to know how our system should act such that ϕ is satisfied. To do this we have to *trick* the model checker. Instead of testing ϕ, we have the model checker test whether or not $\neg\phi$ is satisfied. What this will do is give us a path (typically of the shortest number of cells due to how the model checker iterates is checking method) that actually satisfies ϕ. The straightforward way of using this is to then convert the abstract cells back into the continuous state space as waypoints (e.g. at the cell's centroid) to which you can use a controller to follow.