PROFESSOR: Self-driving cars are pretty cool. It's pretty hot nowadays, so we're going to use this as an example. And in the past we have seen how we can give an autonomous vehicle a goal-- some condition we want it to reach. For example, we might have a car that wants to pick up some passengers along the way, get to some destination, and maybe navigate some worlds.

But let's consider that for something like a self-driving car, we also have a number of other goals or requirements that are sort of implicit along the way, in that we want our self-driving car to obey the rules of the road as it's driving. And these aren't just goal conditions which is a sort of single goal that you're trying to reach, but these are sort of goals that you're trying to maintain throughout the path that you're traveling.

So one example of this is a traffic light. So maybe you guys can help me out with this. Just in plain English, how would you describe the rules of how you would want a car to behave when it comes across a traffic light? Anything? It's very simple. Yes?

AUDIENCE: Slow down when you see yellow, stop when you see red, go if you see green.

PROFESSOR: Yeah. So pretty much stop if you see red, go if you see green, and specifically, if you see red and you stop, you're going to want to stop until you see the green, at which point that sort of condition that you're stopping goes away, and you're able to move on.

And in addition to things like obeying traffic lights, there might also be some other rules of the road that we want to follow. For example, we might want to always stay within the speed limit. And then there might also be some other practical conditions of driving down the road, which is that at some point we're going to need to refuel.

And if we consider a car that might be driving for a really long time, one logical statement we could say about the path that this car takes, and the sequence of states that it goes through is that at every point in time we want the plan to have some future state when we are going to visit a gas station. So this isn't just a single goal that we're trying to reach, but over a very long

time, even if we consider the car to be driving an infinite amount of time, at every point in the time, we're going to be thinking ahead that there's going to be some time when we reach a gas station and refuel.

So these types of conditions, things like staying at a red light until it turns green, always staying within a certain speed limit, or always having some path in the future-- some state in the future when we reach a gas station, are things that we can express with a type of logic called temporal logic.

And there are two things that we hope you're going to be able to do by the time you leave this lecture. The first is be able to model these temporally extended goals using a specific type of temporal logic called Linear Temporal Logic, or LTL. And secondly, we're going to hope that you'll be able to actually apply this to planning, create plans with these temporally extended goals, and in addition just sort of this regular planning we've been dealing with, actually incorporating this idea of preferences into your plans. So a preference basically is not a required condition, but a desired condition that you can use to select between alternative paths. So you know, you might have many different ways that you could reach your goal, but we can use some of these temporally extended goals, like maybe you prefer to break as few of the rules of the road as you can, and you can use those a preferences to choose between plans.

So just as an outline, first we're just going to do a little introduction to linear temporal logic. We're going to talk about what this is-- what LTL is, why we want to use it, and we're going to go through the syntax and the semantics of different LTL operators. Then we're going to walk you through some example LTL problems, and actually talk about complications to plans-- how you create plans with these linear temporal logic and temporal goals. Finally we're going to incorporate preferences into this, talk about how you express preferences, and specifically talk about a language called LPP that allows you to plan with temporal logic and preferences.

So now for the introduction to linear temporal logic. Temporal logic at its core is a formalism for specifying properties of a system that vary with time. So these aren't just conditions that are true at single state, which is what we've mostly been dealing with prepositional logic. Right, so we've been saying things like, condition A and B, but not C, are true at a given state, and that's prepositional logic. Temporal logic is sort of a layer on top of that, when we're dealing not just with what's true at a single state, but actually extending over time as the system moves through a sequence of states, and expressing properties on a temporal level.

So you might have a system that can be represented as a state machine, and a [INAUDIBLE], and it can go through a sequence of different states. And while you might be able to represent the whole system like this, if we actually execute the system we're going to get one path through the system. And that's often called a trace of the system, and you can also think of it as a timeline of the system.

So one example timeline of this system is it might start in state A, and then go to state C, and then go to D, and keep looping around in D forever. So that's one trace, or one timeline of the system, and that's really just a sequence of states that it goes through. And in the past we've seen these in some of the problem sets. For example when we were modeling the warp reactor, we had the valves that could transition between open and closed states. It could also be the whole system-- the starship going through different planets, transitioning through a larger set of locations, picking up passengers, dropping them off. So these could be quite complex.

And so as I said, previously in our problems that we've been modeling, we've been going through a system, and we've been searching for a single state that satisfies all of our goal conditions. Maybe we to reach [? Lavinia ?] and save a certain number of people. But with temporal logic, we can actually express goals that are satisfied over a number of states, and this allows us to do some more expressive goals than we could capture just by looking at a single state at the end.

So for example, what if a problem requires some condition to be met until another condition is met? And for example, you know, when you see a red light, that implies that we should stop until we see a green light. And if we look at a timeline, or a trace of one of these systems, that might look something like this. So we're going along through our system, and at one of the states, we see the red light. So that adds this condition that we should stop. And then that condition will hold forever until we see a green light, at which point that condition is dropped, and we can keep moving on. Another example where we might want to use temporal logic is, what if our problem requires a condition to always eventually be met. And this is what I was talking about with the gas station. So we want to be going on forever, and there should always be some point in the future when we do have a state where we're at a gas station. So we might start our system at a certain state. And our plan says that at some point we do reach a gas station, so that's good. But then when we get to the next state, we also want it to be true that from that point on, we're going to reach another gas station, and after that, going on and

on and on into the future.

These are things that you can't express very easily with the types of logic that we've been dealing with before. So one important distinction that we should clear up before we really dive into the syntax of how these linear temporal logics work is the difference between branching time and linear time. They're really two different models of time that we can be working with. There's two different types of temporal logic that exist to express these different types of time.

So linear time is more similar to the world that we live in, unless you're living in some sci-fi movie where time is branching into different realities, and all sorts of crazy stuff is going on. You can think of time as just this linear timeline. So right now, it's one future state that's going to happen for me. And you know, regardless of how many options I have, there's one realization of the time.

And what this lets us do is that it gives us a single path for our system, and we can reason very exactly about the conditions that are met within that path. And this lets us describe what will always happen in a system. For example, we can guarantee that we always stop at a red light until we see a green light, or we always stay within the speed limit.

This is contrasted with a different model of time, called branching time. And in this model of time, at each time instant, we consider all possible futures. So if we have multiple actions that we could perform at a given state, we're going to consider different timelines that branch off for each of those possible behaviors. And this results in alternate courses of time that we can reason about as a set. And instead of thinking about this as always conditions-- so, our path always does this-- we can think about this more in possibilities. And if we're using branching time, we can say, you know, at this state and time, there exists a future path in which we will always stay under the speed limit, or, at this point in time, all our future paths satisfy a certain condition.

So whereas linear temporal logic is more about guaranteeing what will happen on a given path, branching time allows us to reason about what might possibly happen in the system.

So for our purposes with planning, in part because it's more attractable, and in part because it gives us an exact analysis of a given path, we're going to be doing linear time, and for that reason, we use what's called linear temporal logic.

There's a different type of temporal logic. There's one called CTL and CTL* that deal with

branching time. And those are sort of an extension of linear temporal logic. They add a few other operators that allow you to reason over multiple paths. But for our purposes, we're just going to be using the linear time model.

So kind of as a recap, what linear temporal logic involves-- we're using this linear time model. Another important distinction is that we're actually going to be talking about infinite sequences of states. So instead of us just reasoning up until a certain goal condition is met, we're thinking about systems that could theoretically keep transitioning over time, over and over and over into the future. So if you have one of these state machines, you could think, at each time step, it's going to visit the new state. And if you give it enough time, it's just going to generate an infinite sequence of states. And we can actually express properties that can hold over an infinite number of states, not just the discrete number of states that we might usually think of to reach a goal.

And linear temporal logic can actually be modified to work with discrete set of states. It just requires an additional operator. But the ability to reason about infinite sequences of states gives us a lot of power in talking about guarantees for our system, regardless of how much time we're talking about it.

Finally, all the operators we're going to be looking at in our logic system are what we call forward-looking conditions. So when we're given a state we're going to reasoning about what's going to happen in the future for a potentially infinite amount of time in the future.

There's sort of a symmetric version of this logic that reasons about the past. So for example, we can say-- talk about guarantees that something will eventually happen in the future-- there's sort of a symmetric version where we can talk about, given a certain state, we can guarantee that something had eventually happened in the past. But for us in that we're going to focus on planning-- we're typically planning about what to do in the future, and it makes more sense for us to use this future of updating the set for operators.

And so I guess one little nuance here that I sort of pointed out in the previous slide is that because we're using a linear model of time and not a branching model of time, we're going to be expressing properties that happen over a single path of states, and not properties that happen over several path of states.

So finally I'm just going to talk about a couple of the main application for linear temporal logic before Ellie will dive in, and actually talk more about the semantics and syntax of how we use

this logic.

So the first menus is verification and model checking. We can use this linear temporal logic to create guarantees about a system, and formally verify it for a couple different properties. For example, we can talk about the safety of the system. We can say, over the course of this system, we guarantee that it will never enter a state of a certain condition. We could also talk about the maintenance of properties. So over the entire lifetime of this system, we guarantee that a certain condition will always hold.

The other main application for temporal logic, the one that we're going to be focusing on, is for planning. And this planning using these temporally extended goals that I've been talking about, and it actually extends nicely to talking about temporally extended preferences. So this is given that we have a number of different paths that could reach a goal, we can actually look at all those and see which paths satisfy certain conditions over the sequence of states, and use those conditions as a way of choosing which ones we prefer.

So with that, Ellie is now going to talk about some of the syntax of linear temporal logic.

ELLIE: Hi guys. My name is Ellie, and I'm going to be talking about the syntax of LTL.

So an LTL formula is a series of states. And at each state, you can have various propositions than can be either true or false. So for example-- is it OK if I erase this, Steve?

STEVE: Yeah. Go ahead.

ELLIE: So for example you can have a [INAUDIBLE] animation show that has a series of states that goes on to infinity. And at each state you can have propositions that are either true or false. So the traffic could be red could be an example for a proposition. We could be stopped, et cetera. And there's various logical operators that we can apply to these formulas-- these sets of states, and we can determine if these logical operators are either true or false about each set of states-- each formula. So there's the not operator, the or operator, the and, the implies, if and only and, and like I said, the state-- each proposition of [INAUDIBLE] can either be true or false.

So just to go through some examples of these, the logical operator true says that-- essentially what the operator true says is it's not saying, oh, this state is true. The operator just true says that there are states for all infinity. So true is always true. So no matter what you had for your

state here, the logical operator true holds. However, when you talk about the logical operator true with respect to a certain state or a certain proposition, then it can either be true or false.

So for example, here, the logical operator red light is true at the first state. If this was green, then the logical operator red would be false at the first state. Does that make sense? All right.

One important thing to note about the logical o operators is that they just apply to the very first state in your statement. So when we're discussing logical operators, we haven't gotten to the point where we can express temporal information about our sequence of states.

The next operator is, not. So as you can see here, the traffic light is read at the first state. So the traffic light's not green. Hold, because the proposition green is not true at the first state. And the logical operator, and, at this first state is we're at the red light, and we're getting gas. And so both that we're at the red light and that we're getting gas holds for the first state.

And then the operator, or, essentially says-- I guess a lot of this is probably review for you guys-- logical operators-- but essentially if you have red or green then you could either have the traffic light be red or the traffic light be green at the first state. And as long as one of those two hold, or both of them hold, then it's true. And one thing to note about or lies in if and only if these can be re-written with just the logical operators, and, and, not.

So another way of expressing red and green is to say that you can't have the scenario where both is not red, and it's not green. And I'm not going to go over that for, implies, and, if and only if, just for the sake of time.

So that's just a review of the logical operators, and now we're going to go into the temporal operators.

So when we're talking about our autonomous car, what are some useful temporal operators? Because up until now, we just are describing of the first state in our sequence, which isn't really useful if we want to talk about the entire set of sequences [INAUDIBLE] states.

So what do you guys think are some mutual things that we want to be able to describe? I know Ben touched on some of them at the beginning, but if you guys could recount some of them.

Yes.

AUDIENCE: State connection.

ELLIE: Mm-hmm. State connections.

So exactly. So if I'm in a state right now, might there be concerns about the next state? Is that kind of like what you're saying? Exactly. Yeah. So next is one of the things that we might be concerned about, the next state.

Are there any other tings you guys can think of?

AUDIENCE: Always or [INAUDIBLE]

ELLIE: Yes, exactly. Always.

So like Ben said, we always want to follow the speed limit. We always want to be under 35 miles an hour, or whatever the speed limit is. Anything else?

AUDIENCE: Until.

ELLIE: Until. Yep. So like he said, you want to be stopped until the light turns green.

Yes.

AUDIENCE: Eventually.

ELLIE: Eventually. Yep. So we eventually want to get gas. And eventually. Any other things? Yes.

AUDIENCE: At this state.

ELLIE: At this state. Yep. And so, at this state, is kind of what the logical operators were doing at the beginning, but I agree that's something we have to be able to express. So that's not a specific temporal operator, but it's included in the logical one, so I'll just write it up here for you.

AUDIENCE: Can I ask a question?

ELLIE: Uh huh.

AUDIENCE: So the logical operators, are they acting on the-- so they're acting for a state? They're not acting on the entire chain.

ELLIE: They act for a state. And so if you just wrote the logical operators, just by the definition of LTL, it means let's just analyze the first state and see if it's true. And so it won't tell us anything about the third state or the fourth state or the second state, which is why we have to look at

the next state. So yeah.

Any other questions? OK.

So like you said, next is an important one. So right here we consider OK, the next light will be green. And until is the one that she mentioned, so the light will be red until it becomes green eventually. So the light will eventually at some point in the future turn green. So you might be waiting at the red light for a long time, but eventually it's going to turn green for us. The light will always be red, so I guess you're stuck at the traffic light for a long time, or globally-- that's another name for always.

And lastly, this on is a little bit less intuitive. So what this is saying is the light will always be red until another circumstance coincides with the same state. So the light will be always red until the car gets gas. And then after it gets its gas, then the red is released, and the light can be either green or red after that. So you have red red red red red until we have both red and gas. And then it can be green or it can be red [INAUDIBLE] it's not constrained to being red.

Does that make sense? That one's a little bit less intuitive, so-- is that good? OK.

All right. So now that we've gone over just the intuitive understanding of that, I'm going to give you the actual syntax for that. So x is representative of next. So in the first state, we don't care if p is true. p could be not true, p could be true. It doesn't matter. But in the second state, the next one up in the first state, p, has to hold in order for xp to be a true statement about our sequence of states.

So what do you guys think if I wrote this? What would that mean?

AUDIENCE: Different state [INAUDIBLE]

ELLIE: Yep. So this one over here?

AUDIENCE: Sure. [INAUDIBLE]

ELLIE: Yeah. Perfect.

All right. The until operator. The until operator essentially says, like Ben said, we're stopped until the light is green.

So if you have p until w, you just p has to be true at every state until w, and after that point, p

can be whatever it wants to be.

p eventually, or future operator, says that at some point in the future we will get gas. However it's important to note-- I know when we talked about getting gas we said, oh, we have to always get gas at some point in the future, but the future eventually operator is just at one point in the future. So once you get gas, you don't have to get it again.

Yes.

AUDIENCE: On the previous slide you had a not at the bottom that said that w is required to become true. Does that mean an until operator also implies a future operator on w?

ELLIE: Yeah.

AUDIENCE: All right. [INAUDIBLE]

ELLIE: Yep. But it's the added condition that p will be true until the w.

AUDIENCE: But like whenever you use until, you say p until w that you'd also have to add at some point in the future [INAUDIBLE]

ELLIE: Yeah. Yep, you're right.

AUDIENCE: Well, I mean, it would be possible that this stays p, right?

ELLIE: It could. Yeah. But then if your system-- that would be totally fine if you had your system just be p, but then the operator p until w wouldn't hold unless you had a w at some point in that sequence of p's.

Does that make sense.

AUDIENCE: Yeah. [INAUDIBLE] I guess it does if that's the way to do it, but it doesn't make sense [INAUDIBLE] it.

ELLIE: So you're saying that we could have a bunch of p's, right? Is that what you're saying? And I'm saying that this is a fine state to have. You can have whatever you want. And we're just analyzing if a sentence is true about it.

So this statement, it's not true-- or, sorry. I'm saying that it's not true that we have p and lw because there's no w in our state. But if we add in a w somewhere and we keep all the p's,

then it's true because we have p up until we get to a w. Did that make sense?

AUDIENCE: It made sense that that's the rule, but it doesn't make sense to me why. Because the statement [INAUDIBLE] true and if w never appears. You have p until w, which never happened.

ELLIE: Yeah. If w never happened, then it wouldn't be true.

AUDIENCE: I mean, technically, for the English language it would be true.

AUDIENCE: I guess it's not--

AUDIENCE: If you have p out to infinity, and you never have w's. p until w is true.

AUDIENCE: p until anything is true [INAUDIBLE]

AUDIENCE: Like if you say that I'm hungry until I eat, and you never eat, then you will always be hungry.

ELLIE: Oh, I see. So that makes sense in a logical sense, but that's not how it's described in LTL.

PROFESSOR: There's also an operator that's sort of unofficial called the weak until, and that's sort of saying p until w, or always p. So that kind of gets rid of the condition that it reaches w at some point.

AUDIENCE: It aligns more with the English, too.

PROFESSOR: Yeah.

ELLIE: Yeah.

AUDIENCE: But just the until operator on its own is just sort of like a strong until.

ELLIE: Yeah.

PROFESSOR: So it adds the--

AUDIENCE: Yeah, that's fine.

ELLIE: OK.

PROFESSOR: This is the way it works.

ELLIE: Yeah. Sometimes-- yeah, I get confused about this sometimes. I'll be like, why they

necessarily [? chose any of these, ?] but, OK.

And then the globally one, which we said-- this is the one that you were just describing here [INAUDIBLE] when you were talking about the p at every single state in the future. And then the release operator is you have to have p until you have w, but p and w also have to share the same state when they switch from-- so in until, you don't have to have the p here, but with release, you have to have the p and w occur in the same state. Does that make sense? Yes? OK.

So just like before, operators can be described using not and and. The future, released, and the globally operators can also be described using other temporal operators. And so just as an exercise, can any of you guys tell me which-- these are not in order-- which of these line up with the other-- like, whichever ones they line up with. And I'll give you guys like three minutes to just think about it, and then we'll work through it together on the board.

All right, so have any of you guys thought about which ones might correlate to which ones? Does anyone have any suggestions? Yes.

AUDIENCE: The first one's the top one.

ELLIE: This one is the top one. That's exactly right. So let's continue this state again. So the statement, true, doesn't say anything about what the propositions are. It's just something that's true for all states, right? So if we're saying, true until pi, that means that you can have whatever you want until p occurs.

So does that make sense that, at some point in the future, p has to occur, exactly because we had the strong until, right. Does that make sense to everyone? You guys want to see some [INAUDIBLE]?

All right and now what about this one? We have not as not p. So this is saying that at some point, if you can't get a p set at some point in the future, you don't have p. So what do you think that would correspond to? If at some point in the future, you can't have the situation where there's not p.

AUDIENCE: The third one.

ELLIE: The third one? [INAUDIBLE] So that would be if you have p at every single state, then you

would never have a situation where you don't have p, right?

And so that leaves this last one. You don't have a scenario where you have not p until not w. So if you have not-- OK. It's still a little hard. But I'll come back to that one at the end, OK? But it's just important to remember that you can describe these operators with similar to them so that your planner doesn't have to deal with the global operator or these operators-- eventually operator, it can have less [INAUDIBLE] to deal with.

So yeah, you guys were right. Perfect. And then this is just a recap of the different operators and the other ways that you can express them.

And so there's a few more really cool things that you can do combining these different operators, and the first is that-- remember when Ben was describing that we need to get gas? And we need to get it in the future, but even after we get gas, we're going to have to get it again some time in the future. So it's important that we can describe something that happens infinitely often. And the way that we describe that is saying that we always have the scenario where you will eventually reach p. So I'm trying to think of another way to describe that, but does that make sense that right now in the future, I'm going to get gas. And even after I get that gas, I will eventually in the future also need to get gas again, because your car will always be needing to get gas. Does that make sense? Cool.

And then eventually forever is another scenario that's important to describe. So let's say that the traffic light will eventually turn green forever. We need to be able to say that at some point in the future, although we don't know when, the traffic light will turn green.

So the future operator says that at some point in the future, you will have something come true. And so at this state, p becomes true, but the global operators that happened from this state forever afterwards, which is saying that you eventually will always have this state B be true.

Do you guys have any questions on that? Did I explain that good enough.

AUDIENCE: It seems weird to me that the gas tank is becoming this occasional temporal operator instead of just like the state that we actually have gas, would just be the logical thing that [INAUDIBLE]

ELLIE: Oh, yeah, I--

AUDIENCE: Why is the act of getting gas showing up here like that, versus just, you always need quantity

of gas.

ELLIE: That's true. We do always need some quantity of gas, and you would describe that, like you said, just with the always operator, but I guess my action of getting gas is I was thinking of actually driving up to the gas station and filling it up with gas. And if you didn't do that, you wouldn't be able to always have gas. Does that make sense?

AUDIENCE: Right, so I would think that always having gas would make [INAUDIBLE] of you plan [INAUDIBLE]

ELLIE: Yeah, and depending on how you make your model, you definitely could do that, and that definitely would satisfy the condition.

[INTERPOSING VOICES]

AUDIENCE: --the usual way to deal with it?

ELLIE: Well, it depends on the situation. Maybe you [INAUDIBLE] want to say at some point in the future, the light will always be green, because you want to make sure that cars can go through, and there's a difference between red, and some point it will be green in the future.

So it really depends on your model and how-- I think the way you're saying it, where you always want to have gas, and the proposition gas is how much gas you have in your tank, and if you have it or don't, then I think that that would be a really good way to model it, too.

AUDIENCE: Because this way, if you said, I have to get gas. I planned it for next week, but I've run out of gas this week. Do you know what I mean, like, you have to somehow be tracking the quantity of gas also.

ELLIE: I agree. I agree. I was just using it as an example to explain the concept, but I agree that that definitely probably could be better with more work.

AUDIENCE: Well, I think in both cases what it could represent is having gas all the time, or explicitly saying that you're going to be in gas stations eventually-- always eventually. It depends on how-- maybe we'd want to always have gas from a station, and also getting gas from it. So depending on how you want to model it, but I think both will work.

AUDIENCE: So infinite and often, doesn't really care about how frequently--

ELLIE: Yeah, it doesn't care how frequently.

AUDIENCE: But [INAUDIBLE] for instance something that happens every 60 years. My system lifetime server is 30 years, which means something happens only once. So in this case, infinite and often, how can you filter a switch operator could be the same, right?

ELLIE: Can you repeat that again [INAUDIBLE]

AUDIENCE: So for instance something happens every 60 years. My system lifetime goes to 30 years. After 30 years, I have to get a re-up on the system. Then so during this [INAUDIBLE] only one thing happened. And so it happens off into infinity, but in the lifetime of the system, it only happened once, and therefore in this condition, the infinite and often should be set the same as the future, right, in the current state. Something happened in the future, I would say something could never happen again.

ELLIE: Kind of. So when you're talking about a system that's dying after 30 years, the LTL doesn't actual model something that ends. It would model something to infinity, although there is an extension to LTL that can incorporate that into it, having it have an actual end goal state. And I think in that case, then infinitely often would be different, and would-- I don't know exactly how that would find that situation. Do you know, Ben? I know you had kind of--

BEN: So there's something called metric temporal logic, which associates a time scale of each of these operators. So I think you'd have to use that if you were trying to express that you had to always meet those conditions still within your lifetime. You could have some sort of qualifier to these operators to specify an actual time [INAUDIBLE] where that operator [INAUDIBLE]

AUDIENCE: So which would you file for the instance infinitely often [INAUDIBLE] scenario something at a little bit more frequent of a general system by the time, right. Now we have something you know [INAUDIBLE] 100 years ago for years.

BEN: You know, it's important that the LTL we're explaining here is only dealing with infinite states. So I agree, it's a little bit-- you have to think about that process of how do apply something that goes on forever to a real system. It obviously doesn't model correctly.

AUDIENCE: There has to be a way to-- I mean the future thing, if he has a system that lasts 30 years, and you say future p, somewhere in your design of the system is going to say that the constraint on future p is that it has to happen within 30 years, right? Because future p here, for getting gas, has to happen before I run out of gas. It's not just-- this representation says it just

happens in the future, but somewhere in your constraint checking, it's going to say, hey, we didn't get gas in this plan soon enough, right?

ELLIE: Yeah.

AUDIENCE: Somehow.

ELLIE: Yes, that's definitely true. And these are just tools to help you model it, but it's definitely not the entire model of your system obviously, and so I agree that we definitely would have to incorporate that in. And for finance systems you have to account for that. And that's why they have the extension of LTL which is the finance systems, which we can certainly drop a link of of the papers, and where we got the information, to the class if you guys would like that.

AUDIENCE: Yeah, I think you guys have done a very good job of answering the questions. The three extensions to LTL that I've found to be the most relevant kinds of things that we're doing is metric temporal logic, which is the first one that you mentioned, also to be able to have it over bounded time, which is related, and you mentioned. And then the last one is to deal with uncertainty, and is then a problem of linear temporal logic.

ELLIE: OK. Cool. All right.

So I meant to have this as two separate slides, but what are some true statements about LTL that you guys can tell me about to look at this, or do, and explain why if you look at it.

OK, so why is the next red true?

AUDIENCE: The next step is slow down.

ELLIE: Yep, exactly. Because the next one is red. Why is it true that in the future, that it's green?

[INAUDIBLE]

Because in the future, it's green. And why is it true that there's red until green?

AUDIENCE: It keeps being red until it's green.

ELLIE: Exactly. So it makes sense that if all of these are true, that this culmination of the and statements between all of them is also true, right? And so you could do that with or as well, or you could do that with-- if you had sequences behind here, you could put a future around this

whole thing, and it would be true that at some point in the future, all of this would hold.

Does that make sense? Yes? OK.

AUDIENCE: Why keep it red there? Your red is the next step.

ELLIE: Yep. Exactly. I was just saying if you had more in the past, and you were considering it from the previous--

All right so now Nadia's going to talk about expressing LTL in PDDL3.

So Ben and Ellie have guided through how expressive LTL formulation is. I'm going to formulate the LTL in a classical planner like PDDL3. Now I'm using PDDL3, which is an composed extension of PDDL2.2, which supports some of the LTL operators.

So these are the basic operators that you can use to express the constraints and goals of your plan. So you have at end, always, sometimes, within, at-most-once, sometime-after, sometime-before, always-within, and hold-during. So again, now here is a numeric controls that you can specify, and the ellipses represents an already-existing [INAUDIBLE] goal. This is a goal description-- [INAUDIBLE]

So some of the operators may look familiar to you, because they can also be used to express the constraint in STN, the Simple Temporal Network, that we learned in class. And there are some operators that are unique to LTL or other kinds, like metric temporal network, like always, for example. So now we are going to take a look at how you can express the operators using the PDDL3 language. So we can use, within one occurrence to the next, because there is no explicit next in the PDDL3. And you use always until to express until. And in the future, you use sometimes after, and globally, you can use always. And for release, since it can always be omega or always until omega when you see p. You can use all costs, too, although this [INAUDIBLE].

OK, let's see some examples. So if your goal is to have the traffic light turn red in the next state, you would want to formulate it using within one occurrence, the traffic light would turn red. And let's take a look at a more complicated example. So if you want to model a goal saying that the traffic light would be green until it turns red, at which point it would be red forever.

So this is temporal logic of predicates that express this goal. And then if you want to model

that using PDDL, there is a direct mapping. You can see the direct mapping between the predicates and the PDDL. So this is basically saying, it would always be green until it turns red. And turning red implies it will always turn red.

So next, [? Arleese ?] will tell us how to map between the LTL to PDDL with a Buchi Automata which is a specialized automata.

GUEST SPEAKER: Cool. So I'm [? Arleese. ?] I have bridged the gap between LTL and the planning world. So this is kind of the framework for how you would go from taking a problem with temporally extended goals all the way to a plan.

So as you can kind of imagine from what Ben and Ellie were talking about, LTL is pretty expressive. You can express a lot of different types of goals that include more temporal properties than just a time window for a goal to be completed.

So once we have a kind of defined problem with these temporary extended goals, then you want to model it in a language like LTL or PDDL. So if you have a language like PDDL, and you have a planner that can do an algorithm called progression, which I'll talk about a little later, you can go directly to a plan.

Basically how progression works is it's kind of an algorithm that tells you when you're analyzing a state, and you're analyzing LTL-like formulas that are true in a specific state, how to push formulas that you need to evaluate later on to the next state, and how to keep track of things that you need to continue to evaluate for satisfaction in future states.

Another way you can do this is by taking LTL formulations and translating them into Buchi Automata. So Buchi Automata are basically finite state machines that are extended to handle an infinite sequence of states. And I'll get into more about how Buchi Automata work. After you have a Buchi Automata, you can then translate that into PDDL2, which you guys are all familiar with, and then that PDDL2 can be used from the classical planner, and get a plan.

So a little bit more about Buchi Automata. Again, like I said, it's an extension of a finite state machine.

Oh, yes.

AUDIENCE: Why would have a Buchi Automata when you could just use a planner that will go from PDDL3 straight to the plan?

GUEST SPEAKER: I guess it just depends on how-- what kind of planners we have access to, what other systems you have. It's done both ways. So PDDL3 is still kind of new, and so there's not a lot of planners that actually have progression and can handle PDDL3.

AUDIENCE: And it's also the case that the particular algorithms that are trying to either be verification or planning, maybe exploiting particular properties of the Buchi Automaton, as opposed to the properties of the native language.

GUEST SPEAKER: So you guys are all familiar with regular state machines. Buchi Automata has a very similar formulation. Some slight differences-- you have a set of states, you have an initial state, and you have a transition relation, and then you have a set of accepting states. These accepting states are essentially what replaces finite states in the state machine. We also have a set of symbols that are again simply the transitions between states.

So what an accepting state is, is a Buchi Automata can only be valid if the sequence of transitions visits an accepting state an infinite amount of times.

So I'll get into a couple examples. So let's say you want to model the ticking-tocking of a clock. This is just a regular finite state machine. As you can see, after some number of ticks and tocks, you get into S2, and you're in S2 because you can be in transition only [INAUDIBLE] So you're in S2 basically forever once you get to S2. So we can model this as a Buchi Automata by making S2 an accepted state. This example, making it a Buchi Automata, doesn't really do anything for you. It's just kind of to illustrate what the accepting state does. So the accepting words are a sequence of transitions. In this case it would be an infinite combination of ticks and tocks that would make this Buchi Automata valid. Does that make sense. Questions about that? Here's another example, for example, if I have changed what my accepting state was. So if our accepting state was S1, I now have to visit S1 an infinite amount of times for this Buchi Automata to be valid. That means the only valid sequence of transitions I could have is tock tick tick tick tick for an infinite amount of times. So you can kind of represent different things and different sequences of inputs you might want to have based on which accepting state you use.

This is another example. If you wanted your clock to be tick tock tick tock tick tock, you can have more than one accepting state, and now I have to visit S0 and S1 an infinite amount of times if I want this Buchi Automata to be valid. So then the sequence of transitions I get is tick tock tick tock tick tock.

Does anyone have any questions to really think about in general with Buchi Automata? OK.

It might help to see how we model LTL as Buchi Automata. So here's an example of of these LTL formulas modeled as a Buchi Automata. I guess take a few minutes and see if you can get which one this one might be.

AUDIENCE: [INAUDIBLE]

GUEST SPEAKER: Yeah. Just so that everyone can kind of see that, and future events, we remember that means that eventually p has to be true at least once. So from any sort of input state, you go to S0, and then I do have amount p for some amount of time, but as soon as I have p, I have to get to p, because I have to be in my subject state an infinite amount of times. So I have to execute p at some point to get to my accepting state for this Buchi Automata to be valid. And then once I'm at S1, I can have any other amount of inputs. I'll always be in S1, and that's what the true label means.

AUDIENCE: And it's also something stronger, right, which is you could-- it's eventually globally?

GUEST SPEAKER: Right. Yeah, so actually in this case, after it gets here, my input could be anything. So this basically just says that for my initial state, I have to execute p at least once to get to this accepting state, and once I'm there, I can be there-- I'm always there. Any input will believe me in S1.

AUDIENCE: So just to make this clear for myself and maybe for others, the data string executed that's at R state something that's on the loop. If you go from s0 to s0, that's essentially saying that one of the nodes is not p. If you go from s0 to s1, it's saying that one of the nodes is p.

GUEST SPEAKER: Yeah.

AUDIENCE: If you go from s1 to s1 and take whatever it would be because you set it true.

AUDIENCE: So if you put p where the true is right now, then you you won't have to go through.

PROFESSOR: Right. That's what I missed. By the way, you guys have 25 more minutes.

GUEST SPEAKER: So that's what we talked about. This is future. So the accepted sequence of propositions would be not to not p. Then you have p. Then it can be anything after you get to p.

And then in this case, for Buchi Automata states, which are like [INAUDIBLE], slightly different

than the LTL states we've been talking about. In this case, you're in s0. And then once you get to s1, you're forever in s1.

So globally, looks pretty similar to this. Take a few minutes to think about how I might change this Buchi Automata to [INAUDIBLE] globally after [INAUDIBLE]

AUDIENCE: [INAUDIBLE]

GUEST SPEAKER: Yeah, exactly. So you have initially, your first state by your n because p has to be true forever after you get to this first state. You have an accepting state in p. You can also model not p. If you want to, you can also just have this single see which would encapsulate exactly what Buchi is. This is how you go from LTL to Buchi Automata.

In a real-world scenario, we have multiple LTL formulas all combined. You're Buchi Automata are going to look a little bit more complicated than this. Try multiple accepting states. There's a full weighted model of what an LTL is trying to represent.

AUDIENCE: If we have the not p transition out of there, Like it's possible you can come into s0 to accept the Buchi Automata because it would go to s1, right?

GUEST SPEAKER: Accepting a Buchi Automata after that, it's a little confusing. A Buchi Automata is only valid if the infinite sequence of states satisfies it. If I were to go the p instead of not p, my sequence would accept a transition of p, p, not p. Then that would not be a valid sequence of transition for this Buchi Automata because I'm transitioning out of my accepting state, which I need to have been in for at least the amount of times. And there's no way I can get back to success.

Also, a more defined algorithm [INAUDIBLE] an intuitive way to build a Buchi Automata, or is an algorithm. The other one that's most popularly used was developed by [INAUDIBLE]. This is a pseudocode version of the algorithm. We go from LTL to Buchi.

As you've noticed, it actually uses progression, which is something that planners that take PDDL use to generate plans. An important thing to note, which I won't go into too much detail, this algorithm generates a generalized Buchi Automata, which you then change to a simple Buchi Automata.

The difference is, the generalized Buchi Automata has a set of sets of accepting states. For a generalized Buchi Automata to be accepted, you need to visit an accepting state in each one of those sets of accepting states. At least one of those states has to be listed.

For a simple Buchi Automata, you only have one set of accepting states. And you can visit any one of those for the Buchi Automata to be valid. On the translation between those two is a little bit more complicated. We'll have paper in the preference that will walk through how to do that. But we're not going to do that now.

This is a progression algorithm. Basically, it just tells you how if have a LTL formula f and some current state N-- and usually they involve some time step because we're in the real world and we can't really model infinite amount of time, and we have to make it discrete. So we have some time step, which means successive state.

How do you push certain LTL formulas onto next states to be evaluated later? For example, I'll take this next f as an example. So if f was a next f of some LTL formula, you need to append that formula to the next state to be evaluated in the next state to see if that's going to be true or not.

Similar things for LTL. And I won't go through this [INAUDIBLE] it's here for reference. The next step to this process is going from Buchi Automata, that's a PDDL2. This process is confusing. At least it is confusing for me to understand.

The first thing I'll say is that Buchi states are not equivalent to traditional PDDL states. In a PDDL state, if you have two states, you have the same propositions that are true and false, those states are identical. In the Buchi Automata, that's not necessarily true.

In the Buchi Automata you also have to encapsulate which transitions you can make out of each Buchi state. So s1 and s2 couldn't have the same set of propositions that are [INAUDIBLE] hold. But out of s1-- actually, back up. This is the Buchi Automata for FutureGlobally, which is what [INAUDIBLE] was talking about. So if I enter an f1, the star means I can have any transition that I want in s1. At some point, I make the transition p and whatever the s2, I have to take transition p forever.

However, if I get a sequence of inputs, let's say I get p and then b, and p again, and then p again, it's not clear if I'm in s1 or s2 because in s1 I can execute any kind of state I want. In s1 I can execute any transition I want in s1. 2 I have to execute only p.

So if I got a sequence of p's, I could be in either state. You need something else to activate PDDL2 to basically to determine which state you're and which transitions you can make from

that state. Does that make sense to everyone?

So there's two ways we can transform PDDL2 to encapsulate what a Buchi Automata encapsulates. The first thing is you can create new actions that can encapsulate the allowable transitions of each state. So basically, for every action you have in your traditional PDDL, you have create a repetitive action for each one of these states so you know exactly which transitions you can take out of that state.

This can get long because you have to make a new action for every single one of these states for every action that you want to take. That is one way you can do it.

Another way you can do it is introducing derived predicates. Derived predicates are predicates that don't depend the actions at all and just depend on some other formula in your plan. For example, we can have a derived predicate that explicitly tells you whether you're in s1 or whether you're in s2.

And you can use these predicates in your actions in PDDL2 to make sure, even though which state, x1 or x2, you're in in your Buchi Automata. And then, in your formulation of your plan, you set your final state to be one of the accepting states. I guess that answers one of the questions, though, how you go from being infinite to finite. The accepting state is usually what you place as your final state.

AUDIENCE: Then in the derived predicates, you're purely in the pre-conditions? Or do they also appear in the effects.

GUEST SPEAKER: They also appear in the effects because you need to know how to transition out of s1 to s2. So you'd likely have a derived predicate for both s1 and a derived predicate for s2. And that is also a lot of work to add to the PDDL2 problem.

But it's definitely shorter than the first way. And it definitely encapsulates everything, by the Buchi Automata [INAUDIBLE] Does anyone have any questions about that?

AUDIENCE: So you're changing the action then?

GUEST SPEAKER: Yeah. You're changing the action but you don't need make a new action for every state. In the first formulation, let's say I have some traditional action, like move blue block, or something. I have to create a move block action for each one of these states because I need to know exactly which transitions I can take out of that action.

AUDIENCE: In the second example?

GUEST SPEAKER: And in the second one I can just have one move block. But in my move block I'll have a derived predicate that tells me which state I'm in when I execute that action. The number of actions you have to write is less because you explicitly have predicates that are telling you which state that you're in.

Now I'm going to hand it over to-- if there are no more questions-- hand it over to Mark. And he's going to talk about preferences.

MARK: We're getting a little low on time. Now I'll transition to talking about-- we talked about temporally extended goals. We said that goals are things that always have to be true.

So now let's talk about preferences. Preferences are things that don't necessarily have to be true, but are things that you'd like to be true.

In the classical planning problem, we can formulate it like that. We can set a state S, set a state s0, a set of operators, and a set of goal states. The problem is transition from the initial states and any state. But those are goal states using the operators [INAUDIBLE]

Preference based planning problem introduces a traditional field R, which is a partial or total relation expressing preferences between plans. And this is a little preference symbol right there. That tells you, if there are multiple plans you need to accomplish your goal, which one do you want to use. And again, preferences are properties that are desired but not necessarily required.

There are two [INAUDIBLE] different types of preference languages, quantitative and qualitative. As you would expect, in quantitative languages, we actually assign a numeric value to the different plans in order to compare them. So these plans are a weight of four, and a weight of two, and that tells you which one to prefer. Here are some languages that do that.

For qualitative languages, they actually just have a plan with this property is preferred to this other plan with this property. But we don't actually know any information that knows how much we prefer one to the other, or something like that. An important element of this is when you have quantitative languages, they're totally comparable. Any two sets of plans you can determine which one is preferred.

Whereas in qualitative languages, you could have situations where plan one is preferred to

plan two, and plan one is also preferred to plan 3, but that doesn't give you any information about whether plan two is preferred to plan three. So it's a little bit less expressive [INAUDIBLE].

To go into how you actually express preferences in PDDL3, like we talked about temporally extended goals, there is a PDDL3 syntax to do this. There's a preference label here that you can put on fluents to represent things you prefer. And then there's a function call is-violated that essentially returns the number of times that any fluent that has a preference label was not satisfied in your plan.

For example, if you have a preference, "Traffic light is green until it turns red." Here's our PDDL3 template. It extended this kid's preference, in which we label it with a preference label here. And this is just a name. And then or plan tries to minimize the number of times that this preference is violated. That's one way to express preferences directly in PDDL3.

So now we talk about LDP. LDP is a different language that is quite expressive in terms of types of preferences you can represent. It is a quantitative language, which means we're going to have weights for each of our plans to express our preferences.

We can actually express the strength of the preference. So Goal A is preferred twice or three times as much as Goal B. It's an extension of an older language named PP. Here is the paper if you want to actually check out these details.

The formulas are constructed hierarchically. I'll go through quickly how we actually construct these preference formulas. The lowest level is called a Basic Design Formula, or BDF. That just expresses our temporally extended proposition. So this is just a straight up LTL, basically, that we saw in the first section of the presentation. I'm going to use that I'm cooking dinner example for these slides.

In this case, we always use the future operators. At some point, I might want to cook this program plan. Maybe I want to order takeout to eat dinner.

And then I have some eating spaghetti or eating pizza. Those are two different things, two different options that I could have in my plan.

I don't have to have either of those. Those are just options that I could have. That's the lowest level.

The next level is called Atomic Preference Formulas, or APFs. There are where we express our preferences between the BDFs that we formed in the previous slide. So in this example I'm using weight, specifically to represent preference, with lower weight being preferred.

Here we're saying, we prefer to cook over ordering takeout. This is you where you have to cook, this is where you have to order takeout. And we apply weights to those two expressions.

The first one's preferred, and how much it is preferred over this plan. And here is you prefer to eat spaghetti over eating pizza. So that's the second level.

Third levels called General Purpose Formulas, or GPS. These are where we can do conjunctions or disjunctions, or qualification of our previous formulas. So for example, if we want to say-- because in the previous slides we had two APFs. We had prefer to cook and prefer to eat spaghetti.

And here we can express that we really don't care which one of these we want to satisfy. You can think of this as we actually tried to satisfy APF at the lowest weight. So if we have this "or" operating here, that means that our planner is going to try and satisfy the lowest weight among all these different options here, which means that he doesn't prefer to cook. That's its first goal.

You can also use a handoff rigor, for example. And that's going to minimize the maximum weight against both of those options. Basically, what's it going to try and do-- oops-- what it's going to try and do is minimize the highest weight among these two options. So it's going to try and cook, and then it's going to try and eat spaghetti versus doing these other options. So those are GPFs.

I'm now moving on to Aggregated Preference Formulas, which are the highest level. So with APFs, these define the order in which our different preferences are relaxed. You can, of course, express a lot of different preferences for your planner, but not all of them may be achievable, especially if you're trying to achieve a large number of preferences at the same. You may not actually be able to achieve all of those.

So how do we produce our set in the correct order, in the preferred order so that our plan we end up with isn't even the most preferred plan? That's what APFs like to do. You can express, using this preference operator.

First, I want to try and satisfy both of these in the previous slide. Then if I can't, I want to relax it so I only satisfy G2. And then if I can't do that, then I want to relax it so I only try and satisfy G1. So that gives us the order in which things are relaxed.

It's important here that if you have these situations where you can't distinguish from one another, or you don't really care, we can establish some order. So at the very lowest type of level, we can sort them alphabetically, or something, just to provide some sort of order.

This is just a review of what I've been talking about. BDFs are the lowest level, which express temporally extended propositions. We can apply preference to those using APFs. Then we can combine or join APFs using tell preference formulas. And finally, we can aggregate those preference formulas to determine the order in which you shouldn't relax the propositions to allow yourself to plan it right.

So using this scheme in LPP, we're using both LTL syntax and rules that we talked about in the first section to express temporally extended preferences. The plans that's actually used to solve these types of problems is called P Plan. P Plans can actually handle templates, and the preferences, and a goal at the end, as well.

And it does basically a best first search among all your different options. It's actually one of the planner that uses the left branch. But the one we showed uses progression to take LTL formulas at each point in the plan and determine whether you've satisfied those formulas or not. And if not, it will push them to the next state, so that in the next state you can evaluate whether you've satisfied those formulas. That's how it prunes the search space and tries to always find the most preferred plan to meet your goal.

All right. So that's our presentation. Any questions from anyone?

AUDIENCE: [INAUDIBLE] question. I have a question about [INAUDIBLE] presentation. You guys [INAUDIBLE] and then believes omega. Would you say that omega has to hold if p happens, and they have a conjunction in one state?

ELLIE: Omega doesn't have to hold until we [INAUDIBLE]. Omega just has to hold at the last state of p. Essentially what it's saying is omega releases p.

So once omega happened, p has to happen at the state. And the omega has to release p. Now it happens, so p, you can don't you want. You can be true, you can be false.

AUDIENCE: Sorry. So p has to hold until omega happens.

ELLIE: Yep. And then omega--

[INTERPOSING VOICES]

AUDIENCE: So the definition is just switched to p [INAUDIBLE].

ELLIE: This is an occasion. Occasionally, you have pR omega. But like intuitively--

AUDIENCE: It's just the definition-- oh--

[INTERPOSING VOICES]

AUDIENCE: That's all I was wondering because then I saw that and it was confusing. So that's all. OK, sorry.

ELLIE: That's fine.

MARK: Any other questions?

AUDIENCE: Since there are multiple ways to express the same formula using the grammar, is there any notion of canonical forms, or a least complicated form, or something like that?

MARK: Yeah, there is. Typically they actually-- let me find the slide. So to simplify the algorithms they typically apply the kind of reductions that we showed in the slide to reduce the release and the globally and the future down to just the [INAUDIBLE] of an x. Otherwise, your algorithms going to be able to handle fewer cases. And then they apply the usual rules, trying to push nots out to the left-hand side.

All right. Thanks, everyone.

[APPLAUSE]