Project Report

A Temporal Logic for Modeling Constraints on Software Architecture Evolution

15-816 Modal Logic
Jeffrey Barnes

May 7, 2010

Abstract

My research involves developing models and tools to support architecture-level planning of large software evolutions. Over the past couple of years, we have developed a theoretical framework that we think encapsulates the most important issues in these kinds of evolutions, as well as formal models and tool prototypes within this framework. One important component of our framework involves the formal expression of architectural *constraints* on evolutions. Last year, I devised a simple extension of linear temporal logic (LTL) to support these constraints. This extension aims to increase the expressiveness of LTL without compromising the ease of understanding and writing formulas.

This logic has served us adequately in practice, but we are uncertain of its theoretical properties. In this report, I explore the characteristics of this logic and show that it is on sound theoretical footing. I develop a formal description of the logic and prove a number of important results. I also survey existing work on related logics.

1 Introduction

Architecture evolution—that is, software evolution that effects changes to the overall structure of a system—is a phenomenon common to nearly all software systems of appreciable size. Software systems have to evolve for many reasons—changes in requirements, changes in technology, changes in the business environment—and often these evolutions entail substantial restructuring of the software architecture (i.e., the large-scale structure of the system).

This kind of architecture evolution has far-reaching effects on a system and can take significant time and resources to carry out. Architects must plan such an evolution carefully to avoid expensive mistakes. Unfortunately, there are not presently good architectural models and tools to help these architects develop such plans. Although there is a rich literature on the topic of software evolution, relatively little attention has been directed to the problems specific to architecture evolution. An architectural perspective is essential for evaluating technical decisions with respect to their global impact and their quality and business trade-offs.

There is a rich literature in software architecture as well. But research in software architecture traditionally focuses on tasks like designing a system that meets given goals (functional requirements and quality requirements) subject to certain constraints (which may be technical or business-oriented). This way of thinking about software architecture ignores the evolutionary aspect of what software architects do. In reality, architects need not only a vision of what a system ought to look like, but also a concrete plan for how to achieve it: how to organize the necessary architectural changes given the resources available.

My colleagues and I have developed a framework for understanding architecture evolution, which we view as a first step toward confronting these problems [GBSC09]. One necessity of our model is a way of expressing *constraints* over how a software system may evolve. As part of our work, we devised a logic to allow us to formalize these constraints. In this report, I attempt to formally describe and evaluate this logic.

TYPOGRAPHICAL CONVENTIONS I use sans serif for names of logics, for example LTL. I add subscripts or suffixes to denote extensions or modifications to these logics. For example, $\mathsf{LTL}_{\mathsf{Ric92}}$ is the variation of LTL that appears in the named reference [Ric92]; LTL−b is LTL without binary connectives. I will use the name $\mathsf{LTL}_{\mathsf{AE}}$ to refer to the constraint logic that is the subject of this report (*AE* for *architecture evolution*).

The precedence within formulas, from highest to lowest, is

1. Unary operators and quantifiers
2. Binary temporal operators
3. Binary propositional connectives, which have their usual precedence relative to each other ($\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$)

This precedence is implicitly assumed throughout this report; it is not made explicit in the formalizations.

1.1 Our Model of Architecture Evolution

Our model of architecture evolution begins with a number of assumptions. First, we assume that the structure of the current system is known. Second, we assume that the desired target architecture is likewise known (or, more generally, there could be set of possible target architectures). Finally, we assume that we have formal architectural representations of the initial and target architectures. (I explain what it means to have a formal architectural representation under the subhead "Architecture Description" below.)

These assumptions are problematic in practice. Many software development teams do not have a firm understanding of how their software systems are structured, let alone a formal representation. And while a software owner may have a keen familiarity with the *problems* of the current system, that does not necessarily mean that there is a specific target architecture in mind to fix these problems. To justify our assumptions, we pass the buck to already well-established subfields of software architecture. If the current architecture is not known, it can be reverse-engineered using a variety of techniques for *architecture reconstruction* [SOV02]. Likewise, a target architecture can be devised via traditional techniques for *architecture design* [HKN+07].

Even assuming we know the initial architecture and the target architecture, there are many possible routes from point A to point B. For example, there may be different orders in which we can complete development tasks, different choices for how we stage development operations, and so on. We therefore consider a set of candidate *evolution paths* that would allow us to evolve progressively from the current architecture to a desired target architecture. (An important part of our work is describing how these candidate paths may be generated and how they fit together, but for the purposes of this discussion we may take them as given.) Concretely, an evolution path is a finite sequence of software architectures, where the first element of the sequence is the current architecture, the last element is a target architecture, and the other elements are intermediate architectures that we would pass through in order to reach the target. In our model, the software architect evaluates a set of plausible candidate paths and selects the path that is optimal in some sense (e.g., lowest cost, earliest completion), subject to some constraints. These constraints over evolution paths turn out to be important in our model of architecture evolution, and they will be the focus of my project.

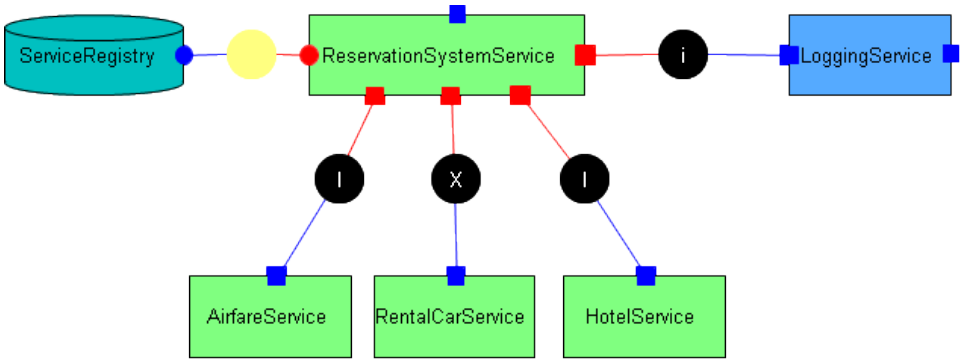

Figure 1: A representation of a simple software architecture in Acme. This is a screen shot of AcmeStudio, a tool for visualizing Acme specifications. (Acme itself is a textual computer language, but most users prefer to work with tools like AcmeStudio that present a graphical interface.) The rectangles and cylinder represent software components; the lines with circles in the middle, connectors. Different colors and shapes represent different element types. In this example, most of the components are instances of type `WebService`, but `ServiceRegistry` is an instance of `Repository`, so it has a different shape and color. Acme provides support for sophisticated rules and constraints; if any were violated, we would see an error indicator.

ARCHITECTURE DESCRIPTION. There are a variety of languages for architecture description [Cle96]. An architecture description language (ADL) is a formal specification language that provides software architects with a means of notating the structure of a software system. Often ADLs are associated with tools that support manipulation and analysis of these architectural models. The ADL we use in our research is Acme [GMW00], but our approach generalizes readily to other architecture description languages.

Acme, like many ADLs, models a software architecture as a graph, where the vertices are software *components* (units of computation) and the edges are software *connectors* (channels of communication). Together, components and connectors are called *architectural elements*. (Acme has a few other kinds of architectural elements besides components and connectors, but they are not as important.) Architectural elements adhere to a type system, so for example a specific component (like `MainCustomerDatabase`) is an instance of a component type (like `Database`). The type system features subtype polymorphism, so `Database` might be a subtype of `DataStore`. Acme provides ways of defining *properties* on architectural elements. For

example, on a connector type that represents network connections, we might define a `bandwidth` property. We can then define *analyses* that may take advantage of these properties and types. In the bandwidth example, we might develop an analysis that calculates the overall latency of the system by using the bandwidth information from the various network connections that appear in the system.

An architecture evolution path is essentially a finite sequence of these architectural models, each of which represents an intermediate architecture. Figure 1 depicts a software architecture. We could imagine it to be an intermediate architecture in an evolution. The next intermediate architecture in the sequence would likely be similar but reflect some changes; perhaps one of the services would be removed, or a new communication bus added.

1.2 Our Path Constraint Language

We previously developed a language to capture path constraints [GBSC09], which we have been using for some time now, but we have not made a serious effort to characterize the formal properties of the logic. Indeed, since my colleagues and I specialize in software engineering rather than logic, we have not been entirely certain that our language makes good theoretical sense. In this project, I have attempted to remedy this.

OUR CONSTRAINT LANGUAGE. Path constraints are predicates over evolution paths that represent things that must be true for a path to be considered valid. Here are some examples from hypothetical evolution projects:

1. The billing component must not be removed until a controller component is present.

2. All the functionality that is present at a release point must remain present throughout the evolution. (That is, once a functionality has been released, it must not be subsequently removed.)

3. The entire evolution must be completed within three years.

We wanted to formalize such constraints so that they could be checked automatically by a tool. We considered a number of possibilities. One choice that seemed natural was temporal logic; since we were dealing with operations that occurred in discrete stages and evaluating the stages in which certain propositions were true, it was an obvious choice. Linear temporal logic (LTL) turned out to be a concise, intuitive way to express a

variety of interesting constraints. For example, constraint 1 above can be expressed in LTL as

$$\mathit{billingComponentPresent}(\mathit{system})\ \mathcal{U}\ \mathit{controllerPresent}(\mathit{system})$$

(Here, *billingComponentPresent* and *controllerPresent* are predicates over systems. Note that each of them is expressible with respect to a single state; they can be defined as rules in Acme.)

However, LTL is not expressive enough for other constraints. For example, if we try to express constraint 2, we encounter a problem:

$$\Box(\mathit{release} \rightarrow \Box \mathit{hasAllFunc}(\mathit{system}, ?))$$

The problem is that we need to refer back to a previous state, namely the previous release point. That is, we want to replace the question mark with a reference to the previous release state, but LTL provides no way to do so. There is no way to express properties over more than one state. To solve this problem, I introduced a notation to allow us to refer directly to states that we have already seen. In my notation, we can formalize constraint 2 thus:

$$\Box\{x\}(\mathit{release} \rightarrow \Box \mathit{hasAllFunc}(\mathit{system}, x)) \tag{1}$$

The braces "save" the current state to the variable x so that we can refer back to it as such in a subsequent step.

Our vision is that software architects will use this language to specify reusable constraints that can be verified by automated tools. That is, software architects will use these constraints to sanity-check evolution paths that they are considering for their software systems. The constraints, in turn, will be written by experienced architectural experts, who will use them to capture reusable knowledge they have about particular domains. For this to work, our constraint language must have certain properties. First, it has to be comprehensible enough to be learned by software engineers (not necessarily by the *average* software engineer, which would be an unrealistic expectation, but rather by software engineers who have an interest in formal specification). I do not address this at all in this project, but it is one of our main motivations; we believe $\mathsf{LTL}_{\mathsf{AE}}$ is more readable and usable than some alternatives we considered. Second, it has to be amenable to analysis. In particular, it needs to be suited to the problem of checking individual paths.

Since my project proposal, I have refocused my project somewhat to more specifically address this use case. In particular, I decided to spend less time coming up with a deductive system for $\mathsf{LTL}_{\mathsf{AE}}$, which is unlikely

to be of practical use (and unlikely to differ meaningfully from existing axiomatizations of similar logics) and instead address this path-checking problem, which had not previously occurred to me as an interesting problem.

1.3 Related Work

There are two very different classes of related work that are of interest here. First, there have been several other attempts to capture architectural transformations using formal models. Second, there have been a number of logics that are formally similar to ours but were developed for different domains. (In addition to these two classes, there is the broader family of work that is related to architecture evolution in general; for an overview of this literature, see our paper on architecture evolution [GBSC09].)

FORMAL MODELS FOR ARCHITECTURAL CHANGE. Several researchers have proposed models to account for aspects of architectural change. Wermelinger and Fiadeiro [WF02] use category theory to model architectural reconfigurations as graph transformations. Grunske [Gru05] uses hypergraphs to specify architectures and uses them to define automatic architectural refactorings that preserve behavior. Spitznagel and Garlan [SG01] focus on transformation of software connectors to add capabilities to communication between components of a system.

These approaches differ from ours in two main ways. First, they tend to be concerned mainly with evolution correctness, rather than supporting more sophisticated analyses about evolution quality, trade-offs, and so on. Second, a primary focus of our approach (although I mention it only incidentally in this project report) is encapsulating reusable architectural expertise about particular domains. The other formal approaches I have cited do not support specialization for specific classes of evolution.

SIMILAR LOGICS. Turning away from the topic of software architecture, we find a number of logics that are similar to ours, particularly with respect to our introduction into LTL of variables that retain their values across states, allowing comparisons across different states of a temporal model.

A logic that is very similar to ours is one developed by [Ric92] to support lists in an object-oriented data model. Richardson's logic is essentially LTL with the addition of "rigid variables," which are semantically identical to our extension to LTL. Richardson's logic does not appear to have been studied for its theoretical properties; the paper has not been widely cited outside the

database community. "Rigid variables" appear elsewhere in the literature as well, most famously Lamport's temporal logic of actions [Lam94], although Lamport's rigid variables are somewhat different.

Another related logic has been developed to model real-time systems. A natural way to specify real-time systems is with LTL, but one problem that arises is the incorporation of hard real-time requirements. Alur and Henzinger [AH89] developed a logic that they called timed propositional temporal logic (TPTL), whose main feature was the introduction of *freeze quantifiers*, which bind a variable to a particular time so that it can be accessed later. These are similar to our rigid variables. There are a couple of differences, which I describe under the next subhead. Henzinger [Hen90] provides an axiomatization of TPTL.

Yet another related logic is Goranko's temporal logic with reference pointers [Gor94]. This logic is quite different in a couple of ways. First, unlike Richardson, Alur and Henzinger, and me, who were devising specification languages for particular domains (object-oriented data models, real-time systems, and software architecture, respectively), Goranko is philosophically motivated. He notes that LTL lacks a way to refer to particular points in time—to express the concept "then." Unlike the other logics we have seen, which give explicit names to states, Goranko's logic simply uses the symbol $\downarrow$ to indicate a point that we might refer to later, then uses $\uparrow$ to refer to it (to say "then"). Syntactically, $\uparrow$ behaves like a propositional variable; semantically, $\uparrow$ is true if the current time is the same as the time of $\downarrow$. Goranko uses this to express things like "now will not occur again:" $\downarrow\Box\neg\uparrow$. So this logic is quite different from ours but shares a similar goal.

A final related family of logics is hybrid logic [BT99, Bla00], where states can be referred to via labels called *nominals*. A nominal is an atomic symbol with the special property that it is true at exactly one state. We can also use a nominal a to build *satisfaction statements*, which have the form $@_a\phi$, which means "ϕ is true relative to the state characterized by a." Finally, often hybrid logics are supplemented with a $\downarrow$ binder, which binds a label to the current state, much like our rigid variables (or a named version of Goranko's $\downarrow$). The result is powerful. For example, we can now define *until* in terms of these hybrid logic constructs:

$$\phi \text{ until } \psi \quad := \quad \downarrow x(\Diamond\downarrow y(\psi \wedge @_x\Box(\Diamond y \rightarrow \phi)))$$

Here, we bind the current state as x and specify that eventually ψ holds; we bind that eventual time to y. Then, with a satisfaction statement, we jump back to x and say that ϕ holds whenever we have yet to see y. Hybrid logics

are rather different from our logic, but the basic idea of named states, and in particular the $\downarrow$ binder, are closely related.

This is not an exhaustive list; Blackburn and Tzakova [BT99], for example, cite a few others, observing, "Labeling the here-and-now seems to be an important operation." Indeed, the idea seems to have been reinvented numerous times, often (as in my case) in ignorance of related work.

HOW $\mathsf{LTL}_{\mathsf{AE}}$ DIFFERS. Our path constraint language, $\mathsf{LTL}_{\mathsf{AE}}$, certainly fits comfortably within this broad family of related logics. Operators that bind variables are nothing new. There are three ways that I think my work can add to this rich literature. First, there are some ways in which $\mathsf{LTL}_{\mathsf{AE}}$ distinguishes itself semantically from its cousins. Although these distinctions are subtle, they turn out to have, in some cases, major theoretical consequences. Second, although the existing literature is rich, it is also somewhat patchy. There are interesting problems that have yet to be tackled. Third, there are some peculiarities of my domain that raise interesting questions. I will now discuss each of these motivations in turn.

First, as I said, there are some differences between $\mathsf{LTL}_{\mathsf{AE}}$ and other logics. I will elaborate for the two related logics that, in my opinion, are the most mature and best-studied: TPTL and hybrid logic.

TPTL was invented by Alur and Henzinger to model real-time systems, but (likely due to their extensive theoretical characterization of their new "freeze quantifier" and their generalization of the idea beyond their domain) their work became quite influential outside of this field. The semantics of TPTL differs from $\mathsf{LTL}_{\mathsf{AE}}$ in two important ways. First, TPTL, like LTL, assumes an infinite sequence of states; our logic assumes a finite sequence. Second, the variables that freeze quantifiers capture are times (natural numbers) rather than architectural models. All they do with the variables they freeze is compare them to other times with operators like $\leq$; we want to do architectural analysis. At first glance, these seem to be peripheral issues, but they turn out to be important. Indeed, Alur and Henzinger themselves showed that small changes to the language can substantially change its theoretical properties; for example, supplementing TPTL with addition over time renders the satisfiability problem highly undecidable.

My results in this project agree with this generalization. These seemingly subtle semantic changes have substantial ramifications on the theoretical properties of the language, and not always in the way that one would expect. For example, I would have predicted that restricting the logic to finite sequences would make almost everything easier. In fact this is not the

case at all. For instance, consider the *expressiveness* problem: the problem of evaluating whether a change to a language would make it more or less expressive. I found that, at least in some cases, proving expressiveness results is substantially *harder* on logics that are restricted to finite sequences. To prove an expressiveness result on an infinite-sequence logic, it often suffices to exhibit two models and demonstrate that the stronger logic can distinguish the two models and the weaker logic cannot. In a finite-sequence logic, it seems that such a simple proof is seldom possible, because almost any logic can distinguish between an arbitrary pair of *finite* models, unless it is incredibly impoverished. Therefore, we have to consider entire (infinite) *classes* of finite models (which brings its own set of challenges) or use more sophisticated proof techniques, even for simple results.

Another example is the problem of *model-checking a path*: evaluating whether a formula holds for a single path. Alur and Henzinger show that this problem is EXPSPACE-complete for TPTL, further noting that it would be undecidable if TPTL were modified to allow more powerful atomic propositions, such as addition over terms. $\mathsf{LTL}_{\mathsf{AE}}$, on the other hand, goes so far as to allow atomic propositions made up of *arbitrary* predicates over arbitrary terms (just like full first-order logic), but then we regain decidability by studying finite sequences rather than infinite sequences. Unsurprisingly, this completely changes the problem of model-checking a path. What is perhaps surprising is that the problem is still hard and interesting. The problem of model-checking a single, finite path might be naively thought to be rather trivial, but Markey and Schnoebelen [MS03] suggest that in fact the problem is of substantial theoretical interest. My results support their contention. As I show in this report, the problem of model-checking a finite $\mathsf{LTL}_{\mathsf{AE}}$ path is hard (PSPACE-complete, which is worse than NP-complete but not as bad as EXPSPACE-complete). Not only that, but this complexity result does not lend itself to easy proof; I had to develop a novel[1] reduction strategy in order to prove PSPACE-hardness.

Much the same can be said of hybrid logic. Although hybrid logic *can* be used to express constraints over finite, linear paths, it does not seem to be done often, at least not often enough that anyone has bothered to study the theoretical properties of that case. And again, although naively we might assume this case to be trivial, boring, or a straightforward specialization of the more general case, in fact interesting and surprising results emerge from these restrictions, such as the expressiveness and model-checking results I just mentioned.

[1] See the caveat in footnote 4 later in the paper.

Hybrid logic is quite different in character and focus from $\mathsf{LTL}_{\mathsf{AE}}$. In particular, although the general idea of *nominals* (propositions that are true in only one state and hence uniquely identify that state) is quite central to hybrid logic, the $\downarrow$ binder that corresponds to our variable-binding operator is not. Rather, $\downarrow$ was a late-breaking addition, imported into hybrid logic from Goranko's temporal logic of reference pointers [Gor94]. (Goranko, in turn, was one of those people who seems to have invented the idea independently, just like Alur and Henzinger in 1989, Richardson in 1992, and me, quite late to the party.) This is not to say that the theory of $\downarrow$ in hybrid logic is underdeveloped—on the contrary, some remarkable results about it have been published—but the focus of hybrid logic is different from ours. More characteristic of hybrid logic than $\downarrow$ is the aforementioned *satisfaction operator*, which I already mentioned; it effectively jumps to a named state. I think this operator is actually quite fascinating, and in future work I would like to consider the ramifications of adding it to $\mathsf{LTL}_{\mathsf{AE}}$. In the meantime, though, not having the satisfaction operator imposes some unexpected challenges. For example, as I note in section 2.2, having the satisfaction operator would make proving the PSPACE-hardness result dramatically easier.

Of course, nominals themselves are also somewhat different from bound variables in $\mathsf{LTL}_{\mathsf{AE}}$. A nominal is a proposition that is true in exactly one state; a bound variable is a *term* that directly captures a state object. This is an important semantic difference, although the effect is much the same.

The second way in which I think there is room to contribute is that the existing literature is in some ways patchy, and there are a number of results that have yet to be obtained. In section 3 I will say more about how this patchiness came to be. But the effect is that there are interesting questions that have not been addressed by any research community, whether the hybrid logic community or those more influenced by Alur and Henzinger or anyone else. The model-checking problem is a good example. The question of the complexity of model-checking a path is a natural one that has been solved for LTL and a number of extensions to LTL [MS03] but not for LTL with a variable-binding operator. I remedy that situation here.

The third distinction that is special to $\mathsf{LTL}_{\mathsf{AE}}$ is the domain to which I am applying it, software architecture evolution. For the most part, this does not make a big impact on my analysis. However, I did have to consider how best to capture constraints on individual architecture models. When we originally envisioned an evolution path constraint language, we freely intermixed temporal operators and Acme predicates. For example, a path

constraint might look like

$$\Box(\exists c : \mathit{Database} \mid \mathit{isOnline}(c))$$

(Here everything but the outer $\Box$ is an Acme rule.) This would have made specification and analysis of the language very difficult. Instead I have taken an approach that separates Acme and $\mathsf{LTL}_{\mathsf{AE}}$. First, Acme is used to define named predicates and functions, which $\mathsf{LTL}_{\mathsf{AE}}$ then treats as black boxes—as ordinary predicates and functions. Thus, we would first specify that the Acme rule

$$\exists c : \mathit{Database} \mid \mathit{isOnline}(c)$$

is a nullary predicate named *someDatabaseOnline*; then our $\mathsf{LTL}_{\mathsf{AE}}$ path constraint becomes simply

$$\Box \mathit{someDatabaseOnline}$$

Not only does this make the language easy to specify and analyze, it also makes formulas much easier to understand. As a side benefit, it makes my project results much more readily generalizable, since there is nothing specific to software architecture in the core language.

2 Project Results

I will now detail the outcome of my study of the properties of our constraint language.

2.1 Formal Descriptions

SYNTAX. Traditionally, the syntax of LTL is defined in a straightforward manner:

Definition 1 (syntax of LTL)**.** Let P be a set of proposition symbols. The formulas ϕ of $\mathsf{LTL}(P)$ (simply LTL for short) are defined inductively:

$$\phi := p \mid \bot \mid \phi_1 \rightarrow \phi_2 \mid \bigcirc \phi \mid \phi_1 \,\mathcal{U}\, \phi_2$$

for $p \in P$.

We can then define the other connectives in terms of these. We are already familiar with how to do this for the propositional connectives: for example, $\neg\phi := \phi \rightarrow \bot$. For the temporal connectives, we define $\Diamond\phi := \top \mathcal{U} \phi$ and $\Box\phi := \neg\Diamond\neg\phi$.

This kind of simple definition will not work for our path constraint language. We need to be concerned not only with simple propositions, but also with predicates and functions. A condition such as "The software architecture has at least one component" could be represent as a proposition p. But a more interesting condition such as "This software architecture has at least the same database components as the one in the previous state" expresses a relation over two different architectures. So in defining a syntax for $\mathsf{LTL}_{\mathsf{AE}}$, we need to be careful to think about what the *atomic formulas* are; proposition symbols alone are inadequate.

The customary way to define the syntax of first-order predicate logic (FOL) is to give separate, inductive definitions for *terms*, *atomic formulas*, and finally formulas, as follows:

Definition 2 (syntax of FOL)**.** Let V be a set of variables. For $n = 0, 1, 2, \ldots$, let F_n be a set of n-ary function symbols and let P_n be a set of n-ary predicate (relation) symbols. Together these sets form a *signature*, which we write as $\Sigma = (V, (F_n)_{n\in\mathbb{N}}, (P_n)_{n\in\mathbb{N}})$. The terms π, atomic formulas α, and formulas ϕ of $\mathsf{FOL}(\Sigma)$ are defined inductively:

$$\begin{aligned}\pi &:= x \mid f(\pi_1, \ldots, \pi_n)\\ \alpha &:= p(\pi_1, \ldots, \pi_n)\\ \phi &:= \alpha \mid \bot \mid \phi_1 \to \phi_2 \mid \forall x \phi\end{aligned}$$

for $x \in V$, $f \in F_n$, and $p \in P_n$.

(Nullary functions are constants, and nullary predicates are propositional variables.)

We can use the same ideas to develop a syntax for $\mathsf{LTL}_{\mathsf{AE}}$.

Definition 3 (syntax of $\mathsf{LTL}_{\mathsf{AE}}$)**.** Let $\Sigma = (V, (F_n)_n, (P_n)_n)$ be a signature as in definition 2. The terms π, atomic formulas α, and formulas ϕ of $\mathsf{LTL}_{\mathsf{AE}}(\Sigma)$ are defined inductively:

$$\begin{aligned}\pi &:= x \mid f(\pi_1, \ldots, \pi_n)\\ \alpha &:= p(\pi_1, \ldots, \pi_n)\\ \phi &:= \alpha \mid \bot \mid \phi_1 \to \phi_2 \mid \bigcirc \phi \mid \phi_1 \,\mathcal{U}\, \phi_2 \mid \{x\}\phi\end{aligned}$$

for $x \in V$, $f \in F_n$, and $p \in P_n$.

To make this discussion more concrete, let us see how we would analyze the example formulas from section 1.2 in terms of this syntax. The first example we saw was

$$\mathit{billingComponentPresent}(\mathit{system}) \,\mathcal{U}\, \mathit{controllerPresent}(\mathit{system})$$

In this example, *billingComponentPresent* and *controllerPresent* are unary predicates. The keyword *system* is a nullary function: it takes no arguments and behaves as a term. (This might be surprising, since I previously said that nullary functions are constants, and *system* does not hold constant—on the contrary, it refers to something different in every state. But in a temporal context, *constant* is not a very good word for a nullary function, because a nullary function can refer to different states depending on the current state, just as a nullary predicate—a proposition—can have a different truth value from state to state. For a different approach, cf. half-order modal logic [Hen90], where functions have a "rigid" interpretation—an interpretation that is independent of state—and predicates have a "flexible" interpretation.) Similarly, in the formula

$$\Box\{s\}(\mathit{release} \rightarrow \Box \mathit{hasAllFunc}(\mathit{system}, s))$$

release is a proposition (nullary predicate) and *hasAllFunc* is a binary predicate.

Where do the predicates and functions come from? Where are things like *controllerPresent* defined? In our model, they are usually defined in something called an *evolution style*, which is a way of encapsulating architectural expertise specific to a domain (e.g. the domain of cloud computing evolutions). The details are not relevant here, but the point is that they are generally not defined in the language, but they are defined in a place where they can be reused appropriately. They can be defined using Acme. For example, an Acme definition of *controllerPresent* would look something like:

controllerPresent(*s* : System) : boolean =
    exists *c* in *s*.components | declaresType(*c*, *Controller*)

Besides these user-defined predicates and functions, there will also be a few predefined in the language. We have already seen one example: the nullary function *system*. Other examples might include a unary *successor* function that returns the next intermediate architecture in the temporal sequence or an *isFinal* predicate that tells whether a state is the final one. However, I do not distinguish between user-defined and predefined symbols in this formalization.

Another question worth thinking about is where in a formula braces are allowed to appear. Definition 3 allows them to appear just about anywhere—around any formula. Thus, a formula like

$$\{x\}\{y\}(p \rightarrow \{z\}q(y, z))$$

is legal, even though it means the same thing as the clearer and simpler $\{x\}(p \to q(x,x))$. We could restrict the syntax so that states can be captured only at certain places within a formula. For example, Alur and Henzinger's TPTL originally did this by defining the formulas of ϕ by

$$\phi := \alpha \mid \mathbf{false} \mid \phi_1 \to \phi_2 \mid \bigcirc x.\phi \mid [x_1.\phi_1]\, \mathcal{U}\, [x_2.\phi_2]$$

so that variables could appear only after $\bigcirc$ or at the beginning of an $\mathcal{U}$ operand [AH89]. (Their terms and atomic formulas are different from ours too, but this is not relevant here.) Incidentally, note that their definition does not allow variables to occur at the beginning of a formula. For them, it is not necessary. Variables just capture time indexes—integers—so to refer to the beginning of the formula, they need not capture the state with a variable; they just use the constant 0.

Anyway, the issue does not matter theoretically. The logic with variables allowed to occur only in these restricted places (i.e., at the beginning of a temporal operand or at the beginning of the entire formula) is equivalent to the logic of definition 3. But allowing variables to occur anywhere makes the syntax and semantics simpler. By 1994, Alur and Henzinger had evidently come to the same conclusion, as the syntax in the journal version of their paper [AH94] looks more like definition 3.

KRIPKE SEMANTICS. We begin by recalling the semantics of LTL. There are various ways to formalize the semantics of LTL. In the following formalization, we identify a state with an interpretation of the proposition symbols. Alternatively we could externalize an interpretation function as a map from states to sets of propositions.

Definition 4 (semantics of LTL)**.** Let P be a set of proposition symbols. Let σ be a sequence of states: $\sigma_1, \sigma_2, \ldots$, where $\sigma_i \subseteq P$ for each i. (Thus, each state comprises the set of proposition symbols that are interpreted to hold true in that state.) We write $\sigma, i \vDash \phi$ to say that σ satisfies the LTL(P) formula ϕ at a time $i > 0$. We define this satisfaction relation inductively:

- $\sigma, i \vDash p$ iff $p \in \sigma_i$ (i.e., iff the propositional letter p is true under the interpretation given by σ_i).
- $\sigma, i \vDash \bot$ never holds.
- $\sigma, i \vDash \phi \to \psi$ iff $\sigma, i \vDash \phi$ implies $\sigma, i \vDash \psi$.
- $\sigma, i \vDash \bigcirc\phi$ iff $\sigma, i+1 \vDash \phi$.

- $\sigma, i \vDash \phi \,\mathcal{U}\, \psi$ iff there is some $j \geq i$ with $\sigma, j \vDash \psi$ such that $\sigma, k \vDash \phi$ whenever $i \leq k < j$.

We have defined other operators, such as $\wedge$ and $\Box$, in terms of these, so I do not present a semantics for them.

There are a number of things we must change to obtain a semantics for $\mathsf{LTL}_{\mathsf{AE}}$. First, LTL normally models a sequence of infinitely many states—$\sigma_1, \sigma_2, \ldots$ in definition 4. We, on the other hand, are interested in expressing constraints over a finite sequence of states: the evolution path, which comprises finitely many intermediate software architectures. So the first thing we need to do to definition 4 is restrict ourselves to a finite sequence of states.

By the way, when dealing with finite sequences of states, there is a question about the meaning of $\bigcirc$. Does $\bigcirc\phi$ hold at the last state in a sequence? Here I will make the arbitrary choice of answering in the affirmative. Thus, $\bigcirc\phi$ means that if there is a next step then ϕ holds there. This is the weak next operator. The strong next operator, $\bar{\bigcirc}\phi$, means that there is a next state and ϕ holds there. Fortunately, we need not add this to our syntax or semantics because $\bar{\bigcirc}$ can be expressed in terms of $\bigcirc$:

$$\bar{\bigcirc}\phi := \neg\bigcirc\neg\phi$$

The second change we need to make is to account for our additions to the syntax. Atomic terms are much richer than they are in LTL. Again we take a cue from FOL. In propositional logic, an interpretation is simply an assignment of truth or falsehood to each proposition symbol. But in FOL, an interpretation is a map that assigns a function to each function symbol and a relation to each predicate symbol. Similarly, in our semantics for $\mathsf{LTL}_{\mathsf{AE}}$, a state now needs to be more than simply an identification of which propositional letters are true; it should be an FOL-style interpretation function that maps the function and predicate symbols of the syntax to functions and relations. (In FOL, these are functions and relations on the domain of quantification; for us they are functions and relations on the temporal states.)

Finally, we need to express the semantics of our new variable-binding operator. This is not as straightforward as it sounds; just adding a line to the definition of the satisfaction relation does not work. We also need a way to keep track of what states the variables are binding to. After making all these changes, we obtain the following semantics for $\mathsf{LTL}_{\mathsf{AE}}$.

Definition 5 (semantics of $\mathsf{LTL}_{\mathsf{AE}}$)**.** Let $\Sigma = (V, (F_n)_n, (P_n)_n)$ be a signature. Let σ be a sequence of states, $\sigma_1, \sigma_2, \ldots, \sigma_n$. As in definition 4, we define a

state to be an interpretation, but now it is more complicated. Each state σ_i is a function that maps each function symbol to a function over states and each predicate symbol to a relation over states. That is,

- If $f : F_n$, then $\sigma_i(f) : S^n \to S$, where S is the set of states.
- If $p : P_n$, then $\sigma_i(p) \subseteq S^n$.

A *variable assignment* $s : V \to S$ is a function that maps variables to states. (We will need this to keep track of what free variables stand for.) We will write $\sigma, i, s \vDash \phi$ to say that K satisfies the $\mathsf{LTL}_{\mathsf{AE}}(\Sigma)$ formula ϕ at time $i \in \{1, \ldots, n\}$ under the assignment s. Before defining the satisfaction relation, however, we need another concept. We define the *denotation* of a term π in the structure σ at time i under assignment s, written $D_{\sigma,i,s}(\pi)$, by

- $D_{\sigma,i,s}(x) := s(x)$
- $D_{\sigma,i,s}(f(\pi_1, \ldots, \pi_n)) := (\sigma_i(f))(D_{\sigma,i,s}(\pi_1), \ldots, D_{\sigma,i,s}(\pi_n))$

Finally, we define the satisfaction relation inductively as follows:

- $\sigma, i, s \vDash p(\pi_1, \ldots, \pi_n)$ iff $(D_{\sigma,i,s}(\pi_1), \ldots, D_{\sigma,i,s}(\pi_n)) \in \sigma_i(p)$
- $\sigma, i, s \vDash \bot$ never holds.
- $\sigma, i, s \vDash \phi \to \psi$ iff $\sigma, i, s \vDash \phi$ implies $\sigma, i, s \vDash \psi$.
- $\sigma, i, s \vDash \bigcirc\phi$ iff $i = n$ or $\sigma, i + 1, s \vDash \phi$.
- $\sigma, i, s \vDash \phi \,\mathcal{U}\, \psi$ iff there is some $j \in \{i, i + 1, \ldots, n\}$ with $\sigma, j, s \vDash \psi$ such that $\sigma, k, s \vDash \phi$ whenever $i \leq k < j$.
- $\sigma, i, s \vDash \{x\}\phi$ iff $\sigma, i, s[x \mapsto \sigma_i] \vDash \phi$ (where $s[x \mapsto \sigma_i]$ is the same assignment as s except with x now assigned to σ_i)

If ϕ is a closed sentence (i.e., has no free variables) then the assignment s is irrelevant and we may simply write $\sigma, i \vDash \phi$.

In the end, we get something that looks more like the semantics of FOL than the semantics of LTL. By now it becomes clear why some authors refer to the variable-binding construct as a kind of quantifier; not only does it look like one, but it also requires a similar formal apparatus for expressing the semantics.

2.2 Model Checking

As a practical matter, the model-checking problem is one of immediate importance to my research, certainly much more so than a deductive system for $\mathsf{LTL}_{\mathsf{AE}}$, appealing though that may be for theoretical reasons. Our primary and most essential use case for these path constraints, after all, is to check whether a given path satisfies a given constraint.

In general, the model-checking problem is the problem of checking whether a specific formula is true of a specific Kripke structure. More specifically, model checking is usually used to verify that a state transition system has some property. For some logics, such as CTL, this is pretty easy. For others, such as LTL, it is hard—PSPACE-complete, in fact. That is, given a finite state transition system, determining whether an LTL formula is valid in that transition system is PSPACE-complete [SC85]. Moreover, the solution to the model-checking problem for LTL is intellectually rather challenging too (much more complicated, for example, than the CTL model-checking algorithm we saw in class). The traditional model-checking algorithm for LTL involves an intricate tableau construction. There is certainly not much hope that our variable-binding construct will make things any easier, especially in light of the result that model-checking TPTL is EXPSPACE-hard (which is worse than PSPACE-complete) [AH89].

Fortunately, we are not terribly interested in this form of the problem. Instead, we are interested in model-checking a single, particular path—not verifying a formula over an entire state transition system. Remember, our primary use case is telling software architects whether the paths that they have planned are admissible according to the constraints. We do not need to check all the paths in some transition system, nor even a great number of paths—just one, or a few, at a time. Likewise, all our paths are finite—and in fact rather short, since they are explicitly defined by humans.

Model-checking a single path is a much easier problem computationally, but one that has been recognized in recent years as theoretically interesting [MS03].[2] For pure, propositional LTL, model-checking a formula of length ℓ on a path of length m takes $O(\ell m)$. The algorithm for single-path model checking for LTL is the same as the familiar dynamic-programming algorithm for model-checking CTL, since LTL and CTL coincide over individual paths. Here is the algorithm:

[2] In the program verification community, the problem of checking a finite, single trace is known as *runtime verification*, with the term *model checking* reserved for checking entire state structures [BLS07]. I will stick with *model checking* for both, because *runtime* means something else in software architecture.

Theorem 1 (model-checking $\mathsf{LTL_f}$ paths [MS03])**.** *Let σ be a temporal sequence of finite length m. Let ϕ be an* LTL *formula of length ℓ. Then there is an algorithm that determines whether $\sigma, i \vDash \phi$ in $O(\ell m)$.*

Proof. Let $\phi_1, \ldots, \phi_n$ be the subformulas of ϕ, listed in order of nondecreasing length. Thus ϕ_n is ϕ itself and ϕ_1 is an atomic formula.

I now show that we can construct a Boolean matrix, $[t_{j,k}]_{m \times n}$, where $t_{j,k} = \top$ iff $\sigma, j \vDash \phi_k$. The proof is by induction. Since ϕ_1 is atomic, we can immediately determine, for $j = 1, \ldots, m$, whether $\sigma, j \vDash \phi_1$. Thus, we can fill the first column of the matrix (in time $O(n)$).

Now, for the inductive step, fix k and suppose we have filled columns 1 through $k - 1$. We now must fill cell $t_{j,k}$ for each j. We split by cases.

Case: ϕ_k has the form $\bigcirc\psi$. Since ψ is a subformula of ϕ_k, it must be one of $\phi_1, \ldots, \phi_{k-1}$; say $\psi = \phi_h$. We now define

$$\begin{aligned} t_{j,k} &:= t_{j+1,h} \qquad \text{for } j = 1, \ldots, m-1 \\ t_{j,m} &:= \top \end{aligned}$$

To fill the entire kth column, we need to make $m - 1$ lookups and m assignments, which takes $O(m)$ time.

Case: ϕ_k has the form $\chi \,\mathcal{U}\, \psi$. Since χ and ψ are subformulas of ϕ_k, we can write $\chi = \phi_g$ and $\psi = \phi_h$ for some $g, h < k$. Now, to fill the kth column in $O(m)$ time, we work in reverse order, filling in $t_{m,k}, \ldots, t_{1,k}$. We can see that $t_{m,k}$ should be true iff $t_{m,h} = \top$. Then, t_{m-1}, k should be true iff either $t_{m-1,h} = \top$, or $t_{a,g} = \top$ and also $t_{m,k}$ (which we just filled in) was true. Here is the pseudocode for this algorithm, which fills the column in $O(m)$ time.

$prev := \bot$
for $j := m$ **to** 1 **step** -1
    $prev := t_{j,h}$ **or** $(t_{j,g}$ **and** $prev)$
    $t_{j,k} := prev$

Case: ϕ_k has the form $\chi \to \psi$. Again write $\chi = \phi_g$ and $\psi = \phi_h$. Define

$$t_{j,k} := \begin{cases} \top & \text{if } t_{j,g} = \bot \text{ or } t_{j,h} = \top \\ \bot & \text{if not} \end{cases}$$

for each j. The whole column takes $O(m)$ time (two lookups per cell).

In any case, we fill the kth row in $O(m)$ time. We can thus fill the entire table in $O(mn)$ time. We read off the answer to $\sigma, j \vDash \phi$ from $t_{j,n}$. □

Things become messier when we add the variable-binding operator, $\{x\}$. This simple dynamic-programming algorithm is no longer adequate, because we also need to keep track of variable valuations. For example, in the formula

$$\Box\{x\}\Box p(x) \tag{2}$$

the truth of $p(x)$ depends not only on the state at which we are evaluating $p(x)$, but also on the value of x. To determine whether formula (2) holds on a path, we ultimately need to evaluate $p(x)$ at each state *and for each value of* x. The subformula $p(x)$ thus needs to be evaluated $O(m^2)$ times; in model-checking finite LTL paths, we never need to evaluate a subformula more than m times. So it is clear that model-checking $\mathsf{LTL}_{\mathsf{AE}}$ paths will be harder than model-checking finite LTL paths. But how hard?

A strong hint that we have departed from the comfortable realm of polynomial-time algorithms comes from extending formula (2), yielding a formula of the form

$$\Box\{x_1\}\Box\{x_2\}\cdots\Box\{x_k\}\Box p(x_1,\ldots,x_k)$$

The only apparent way to check this formula is to evaluate $p(x_1,\ldots,x_k)$ at each state and at each possible valuation of $x_1,\ldots,x_k$. There are $\Theta(m^k)$ such valuations. (Incidentally, you may note that not all m^k valuations are actually relevant. In fact the valuations that need to be checked are exactly those satisfying $x_1 \leq \cdots \leq x_k$. The number of such valuations is the binomial coefficient $\binom{m+k-1}{k}$. If m is fixed, then this is $\Theta(m^k)$.) Note that k, the number of rigid variables, is asymptotically proportional to the length of the formula, ℓ. That is, the formula is made up of alternating □s and variable bindings (plus a finite tail that becomes insignificant as $\ell \to \infty$), so the number of variables grows in proportion to the length of the formula: k is $\Theta(\ell)$. Thus, we have to evaluate $p(x_1,\ldots,x_k)$ under $\Theta(m^\ell)$ different valuations; there is no apparent way to check the formula in polynomial time.

However, we can model-check an $\mathsf{LTL}_{\mathsf{AE}}$ in polynomial *space*, because we only need to work with one valuation at a time. In fact, even the naive recursive algorithm is polynomial-space, as the following theorem shows.

Theorem 2. *There is a polynomial-space algorithm for model-checking an* $\mathsf{LTL}_{\mathsf{AE}}$ *formula on a single path.*

Proof. Let σ, ϕ, ℓ, m be defined as in theorem 1. We define an algorithm as follows:

```
CHECK(φ, j, s)
    if φ is an atomic proposition
        evaluate φ at j using assignment s
    if φ = ⊥
        return ⊥
    if φ is of the form χ → ψ
        return CHECK(ψ, j, s) or not CHECK(χ, j, s)
    if φ is of the form ◯ψ
        return CHECK(ψ, j + 1, s)
    if φ is of the form χ 𝒰 ψ
        for i = j to m
            if CHECK(ψ, i, s)
                return ⊤
            if not CHECK(χ, i, s)
                return ⊥
        return ⊥
    if φ is of the form {x}ψ
        return CHECK(ψ, j, s[x ↦ σ_i])
```

It should be clear that this algorithm correctly evaluates ϕ at j under assignment s; for the most part, it is a direct implementation of the semantics.

What is its space complexity? The stack depth will certainly never exceed the number of subformulas of ϕ, since every recursive call is of a strict subformula. Since the number of subformulas is less than ℓ, this means the stack depth never exceeds ℓ. Within each execution of CHECK (excluding the recursion), we use only $O(1)$ space. Thus, the total space needed for the algorithm is only $O(\ell)$. □

This shows that the $\mathsf{LTL}_{\mathsf{AE}}$ model-checking problem is in PSPACE. I will now prove that it is PSPACE-complete (which is worse than NP-complete). The traditional way to prove that a problem is PSPACE-hard is to prove that the quantified-Boolean-formula (QBF) problem reduces to it. QBF is the canonical PSPACE-complete problem; it fulfills the same role for PSPACE that the Boolean satisfiability problem (SAT) fulfills for NP. In fact QBF is a direct generalization of SAT. A quantified Boolean formula is just what it sounds like: a formula such as

$$\exists x \forall y \exists z (z \wedge x \vee y) \tag{3}$$

where each variable is interpreted as Boolean and is quantified. (This QBF is in prenex normal form: it comprises a string of quantifiers followed by a quantifier-free propositional form. Any QBF can be converted to prenex normal form in polynomial time, so from now on we will assume prenex normal form, as is typical in QBF proofs.) The QBF problem is to determine whether the formula is true. SAT can be viewed as the special case of QBF where all the quantifiers are existential.

There is a fairly obvious transformation from QBF to our model-checking problem, but unfortunately this obvious reduction does not quite work. The obvious reduction is to change the $\forall$s into $\Box$s and the $\exists$s into $\Diamond$s, add variable bindings after each quantifier, then check the formula on a path of length 2, where the first state represents falsehood and the second represents truth. Thus formula (3) would become

$$\Diamond\{x\}\Box\{y\}\Diamond\{z\}(t(z) \wedge t(x) \vee t(y))$$

where t is a predicate that is false when its argument refers to state 1 and true for state 2. This *would* work, if we interpreted $\Box$ to mean "for all states" rather than "for all future states" and likewise for $\Diamond$. But since y cannot be a state previous to x and z cannot be previous to y, there are some assignments that will never arise from this $\mathsf{LTL}_{\mathsf{AE}}$ formula, such as $\{x \mapsto 2, y \mapsto 2, z \mapsto 1\}$. Thus the $\mathsf{LTL}_{\mathsf{AE}}$ formula is not equivalent to formula (3).

However, it is temptingly close, and it is easy to imagine language extensions that would make the proof work. For example, if we had an operator called @ that allowed us to jump back to a previous state, we could translate formula (3) as

$$\{h\}\Diamond\{x\}@_0\Box\{y\}@_h\Diamond\{z\}(t(z) \wedge t(x) \vee t(y))$$

The @ operator exists in hybrid logic, and indeed precisely this approach has been recently used to prove that model-checking a hybrid logic with @ and variable binders (notated $\downarrow$ in hybrid logic) is PSPACE-hard [FdR06].

Proving that model-checking an $\mathsf{LTL}_{\mathsf{AE}}$ formula is PSPACE-hard is more challenging. However, I have been able to develop a reduction of QBF that does the job. Instead of a simple two-state model, I build a $2k$-state model, where k is the number of quantifiers. The odd states will represent falsehood; the even, truth. The following proof provides the details.

Theorem 3. *The problem of model-checking an* $\mathsf{LTL}_{\mathsf{AE}}$ *formula on a single path is PSPACE-complete.*

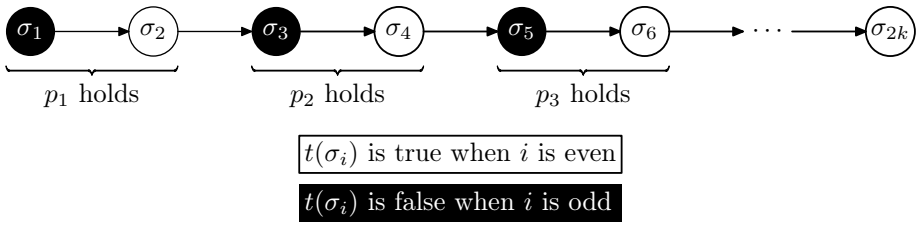

Figure 2: A graphical representation of the $\mathsf{LTL}_{\mathsf{AE}}$ model that we construct in the reduction of a QBF.

Proof. By theorem 2, the problem is in PSPACE. It remains to show that it is PSPACE-hard. I exhibit a polynomial-time reduction of the quantified-Boolean-formula (QBF) problem to the problem of model-checking an $\mathsf{LTL}_{\mathsf{AE}}$ formula.

Let

$$Q_1 x_1 Q_2 x_2 \cdots Q_k x_k \phi(x_1, x_2, \ldots, x_k)$$

be a QBF in prenex normal form. Thus $Q_1, \ldots, Q_k$ are quantifiers and $\phi(x_1, \ldots, x_k)$ is an abbreviation for a propositional Boolean formula over the variables.

I now illustrate how to translate this QBF into an $\mathsf{LTL}_{\mathsf{AE}}$ model-checking problem. Define the signature by

$$\Sigma = ((x_1, \ldots, x_k), (\emptyset), (\{p_1, \ldots, p_k\}, \{t\}, \emptyset, \emptyset, \ldots))$$

Thus $x_1, \ldots, x_k$ will be variables, $p_1, \ldots, p_k$ will be nullary predicates, and t will be a unary predicate.

Now let σ be a temporal sequence of $2k$ states, $\sigma_1, \ldots, \sigma_{2k}$. We now define the interpretation of the nullary predicates at each state by

$$\sigma_i(p_j) = \begin{cases} \top & \text{if } 2j - 1 \leq i \leq 2j \\ \bot & \text{otherwise} \end{cases}$$

for all i, j. (A nullary relation can be simply identified with $\top$ or $\bot$.) We define the interpretation of the unary predicate t by

$$\sigma_i(t) = \{\sigma_2, \sigma_4, \sigma_6, \ldots, \sigma_{2k}\}$$

Figure 2 summarizes this model. Note that we can construct this model in linear time. This is important, since our reduction has to be polynomial-time for the proof to succeed.

We now construct the $\mathsf{LTL}_{\mathsf{AE}}(\Sigma)$ formula that corresponds to the QBF. For the quantifier part, we translate:

- $\forall x_i$ into the string "$\Box(p_i \rightarrow \{x_i\}$"
- $\exists x_i$ into the string "$\Diamond(p_i \wedge \{x_i\}$"

The quantifier-free part we simply transcribe literally, except that each occurrence of x_i becomes $t(x_i)$. At the end of the formula, we add k closing parentheses. For example, formula (3) becomes

$$\Diamond(p \wedge \{x\}\Box(q \rightarrow \{y\}\Diamond(r \wedge \{z\}(t(z) \wedge t(x) \vee t(y)))))$$

This formula, too, can be constructed in linear time. (It is quite a bit larger than the original QBF, but only by a constant factor.)

It remains to show that the constructed $\mathsf{LTL}_{\mathsf{AE}}$ formula (call it ψ) is true iff the original QBF (call it χ) is true. Let χ_i denote χ with the first i quantifiers removed. Similarly let ψ_i denote ψ with the first i quantifier translations (and the last i parentheses) removed. I will show that for any i, for any $j < 2i$, and for any assignment s of the variables,

$$\sigma, j, s \vDash_{\mathsf{LTL}_{\mathsf{AE}}} \psi_i \quad \text{iff} \quad s^* \vDash_{\mathsf{FOL}} \chi_i$$

Note that we translate the $\mathsf{LTL}_{\mathsf{AE}}$ assignment $s : V \rightarrow S$ into the FOL assignment $s^* : V \rightarrow \{\top, \bot\}$ given by

$$s^*(x_j) = \begin{cases} \top & \text{if } s(x_j) \text{ is one of } \sigma_2, \sigma_4, \sigma_6, \ldots, \sigma_{2k} \\ \bot & \text{if not} \end{cases}$$

The proof is by induction on i. The base case is $i = k$, where all the quantifiers have been removed, so χ_i and ψ_i are both propositional formulas. The propositional connectives have the same semantics in $\mathsf{LTL}_{\mathsf{AE}}$ and FOL, so it suffices to show that for any $j < 2i$ and for each h,

$$\sigma, j, s \vDash_{\mathsf{LTL}_{\mathsf{AE}}} t(x_h) \quad \text{iff} \quad s^* \vDash_{\mathsf{FOL}} x_h$$

By the definition of t and the semantics of $\mathsf{LTL}_{\mathsf{AE}}$, it suffices to show that

$$s(x_h) \in \{\sigma_2, \sigma_4, \sigma_6, \ldots, \sigma_{2k}\} \quad \text{iff} \quad s^* \vDash_{\mathsf{FOL}} x_h$$

This is immediate from the definition of s^*.

For the inductive step, suppose we have proven the result for $i + 1$, so we know that for any $j < 2i + 2$ and for any assignment s,

$$\sigma, j, s \vDash_{\mathsf{LTL}_{\mathsf{AE}}} \psi_{i+1} \quad \text{iff} \quad s^* \vDash_{\mathsf{FOL}} \chi_{i+1}$$

We split by cases based on whether the ith quantifier is $\forall$ or $\exists$.

Case: $\forall$. We must show that for each $j < 2i$ and for any s,

$$\sigma, j, s \vDash_{\mathsf{LTL_{AE}}} \Box(p_i \to \{x_i\}\psi_{i+1}) \quad \text{iff} \quad s^* \vDash_{\mathsf{FOL}} \forall x_i \chi_{i+1}$$

Whenever $j < 2i$, observe that

$$\begin{aligned}
& \sigma, j, s \vDash_{\mathsf{LTL_{AE}}} \Box(p_i \to \{x_i\}\psi_{i+1}) \\
\text{iff} \quad & \sigma, h, s \vDash_{\mathsf{LTL_{AE}}} p_i \to \{x_i\}\psi_{i+1} \text{ for all } h \geq j && \text{(df. } \Box\text{)} \\
\text{iff} \quad & \sigma, h, s \vDash_{\mathsf{LTL_{AE}}} \{x_i\}\psi_{i+1} \text{ for all } h \geq j \text{ with } \sigma, h, s \vDash p_i && \text{(df. } \to\text{)} \\
\text{iff} \quad & \sigma, h, s \vDash_{\mathsf{LTL_{AE}}} \{x_i\}\psi_{i+1} \text{ for all } h \geq j \text{ with } 2i - 1 \leq h \leq 2i && \text{(df. } p_i\text{)} \\
\text{iff} \quad & \sigma, h, s \vDash_{\mathsf{LTL_{AE}}} \{x_i\}\psi_{i+1} \text{ for all } h \text{ with } 2i - 1 \leq h \leq 2i && (j < 2i) \\
\text{iff} \quad & \sigma, h, s[x_i \mapsto \sigma_h] \vDash_{\mathsf{LTL_{AE}}} \psi_{i+1} \text{ for all } h \text{ with } 2i - 1 \leq h \leq 2i && \text{(df. } \{\}\text{)}
\end{aligned}$$

Similarly,

$$s^* \vDash_{\mathsf{FOL}} \forall x_i \chi_{i+1} \quad \text{iff} \quad s^*[x_i \mapsto h] \vDash_{\mathsf{FOL}} \chi_{i+1} \text{ for all } h \in \{\top, \bot\}$$

Thus, it suffices to show that

$$\begin{aligned}
& \sigma, h, s[x_i \mapsto \sigma_h] \vDash_{\mathsf{LTL_{AE}}} \psi_{i+1} \text{ for all } h \in \{2i - 1, 2i\} \\
\text{iff} \quad & s^*[x_i \mapsto h] \vDash_{\mathsf{FOL}} \chi_{i+1} \text{ for all } h \in \{\top, \bot\}
\end{aligned}$$

For this, it suffices to show that

$$\begin{aligned}
& \sigma, 2i - 1, s[x_i \mapsto \sigma_{2i-1}] \vDash_{\mathsf{LTL_{AE}}} \psi_{i+1} \quad \text{iff} \quad s^*[x_i \mapsto \bot] \vDash_{\mathsf{FOL}} \chi_{i+1} \\
\text{and} \quad & \sigma, 2i, s[x_i \mapsto \sigma_{2i}] \vDash_{\mathsf{LTL_{AE}}} \psi_{i+1} \quad \text{iff} \quad s^*[x_i \mapsto \top] \vDash_{\mathsf{FOL}} \chi_{i+1}
\end{aligned}$$

But each of these is simply an instance of the induction hypothesis, since by the definition of s^* we have

$$\begin{aligned}
(s[x_i \mapsto \sigma_{2i-1}])^* &= s^*[x_i \mapsto \bot] \\
(s[x_i \mapsto \sigma_{2i}])^* &= s^*[x_i \mapsto \top]
\end{aligned}$$

Case: $\exists$. We must show that for each $j < 2i$ and for any s,

$$\sigma, j, s \vDash_{\mathsf{LTL_{AE}}} \Diamond(p_i \wedge \{x_i\}\psi_{i+1}) \quad \text{iff} \quad s^* \vDash_{\mathsf{FOL}} \exists x_i \chi_{i+1}$$

Whenever $j < 2i$, observe that

$$\begin{aligned}
& \sigma, j, s \vDash_{\mathsf{LTL_{AE}}} \Diamond(p_i \wedge \{x_i\}\psi_{i+1}) \\
\text{iff} \quad & \sigma, h, s \vDash_{\mathsf{LTL_{AE}}} \{x_i\}\psi_{i+1} \text{ and } \sigma, h, s \vDash_{\mathsf{LTL_{AE}}} p_i \text{ for some } h \geq j && \text{(df. } \Diamond\text{,}\wedge\text{)} \\
\text{iff} \quad & \sigma, h, s \vDash_{\mathsf{LTL_{AE}}} \{x_i\}\psi_{i+1} \text{ and } 2i - 1 \leq h \leq 2i \text{ for some } h \geq j && \text{(df. } p_i\text{)} \\
\text{iff} \quad & \sigma, h, s \vDash_{\mathsf{LTL_{AE}}} \{x_i\}\psi_{i+1} \text{ for some } h \text{ with } 2i - 1 \leq h \leq 2i && (j < 2i) \\
\text{iff} \quad & \sigma, h, s[x_i \mapsto \sigma_h] \vDash_{\mathsf{LTL_{AE}}} \psi_{i+1} \text{ for some } h \text{ with } 2i - 1 \leq h \leq 2i && \text{(df. } \{\}\text{)}
\end{aligned}$$

Similarly,

$$s^* \vDash_{\mathsf{FOL}} \exists x_i \chi_{i+1} \quad \text{iff} \quad s^*[x_i \mapsto h] \vDash_{\mathsf{FOL}} \chi_{i+1} \text{ for some } h \in \{\top, \bot\}$$

Thus, it suffices to show that

$$\begin{aligned} & \sigma, h, s[x_i \mapsto \sigma_h] \vDash_{\mathsf{LTL_{AE}}} \psi_{i+1} \text{ for some } h \in \{2i-1, 2i\} \\ \text{iff} \quad & s^*[x_i \mapsto h] \vDash_{\mathsf{FOL}} \chi_{i+1} \text{ for some } h \in \{\top, \bot\} \end{aligned}$$

For this, it suffices to show that

$$\begin{aligned} & \sigma, 2i-1, s[x_i \mapsto \sigma_{2i-1}] \vDash_{\mathsf{LTL_{AE}}} \psi_{i+1} \quad \text{iff} \quad s^*[x_i \mapsto \bot] \vDash_{\mathsf{FOL}} \chi_{i+1} \\ \text{and} \quad & \sigma, 2i, s[x_i \mapsto \sigma_{2i}] \vDash_{\mathsf{LTL_{AE}}} \psi_{i+1} \quad \text{iff} \quad s^*[x_i \mapsto \top] \vDash_{\mathsf{FOL}} \chi_{i+1} \end{aligned}$$

As in the previous case, these follow from the induction hypothesis.

This concludes the inductive proof that for any i, any $j < 2i$, and any s,

$$\sigma, j, s \vDash_{\mathsf{LTL_{AE}}} \psi_i \quad \text{iff} \quad s^* \vDash_{\mathsf{FOL}} \chi_i$$

Choosing $i = 0$ (so that $\chi_i = \chi$ and $\psi_i = \psi$), $j = 1$ (the initial state), and s to be arbitrary, we obtain

$$\sigma, 1 \vDash_{\mathsf{LTL_{AE}}} \psi \quad \text{iff} \quad \vDash_{\mathsf{FOL}} \chi$$

as desired. □

So model-checking an $\mathsf{LTL_{AE}}$ formula is PSPACE-complete. This is somewhat disappointing, as PSPACE-complete problems are considered intractable (worse than NP-complete). On the other hand, it is perhaps not too surprising, since, as we have seen, the variable-binding extension is a sort of quantifier, and a great many interesting decision problems for quantified logic are PSPACE-complete [GJ79], including even the problem of checking first-order monadic logic over finite paths [MS03]. And of course hybrid logic with $\downarrow$ has the same problem as $\mathsf{LTL_{AE}}$ (and, as I mentioned, the proof of PSPACE-completeness is quite easy for hybrid logic with $\downarrow$ and @).

LTL itself, though linear-time for the problem of model-checking individual paths, is well-known to be PSPACE-complete for the problem of model-checking entire state systems [SC85]. Indeed, we can count ourselves fortunate that the single-finite-path problem is the one we are interested in. Alur and Henzinger [AH89], who developed TPTL, were interested in the harder problem of checking a formula on a state system and found it to

be EXPSPACE-complete (which is worse than PSPACE-complete)—which for them was a successful outcome because it was not clear that the problem was even decidable. We would undoubtedly face similar difficulties in attempting a solution for $\mathsf{LTL}_{\mathsf{AE}}$.

Anyway, there are practical reasons not to be terribly concerned about the PSPACE-hardness of checking $\mathsf{LTL}_{\mathsf{AE}}$ paths. First, most formulas we encounter in practice are not likely to be terribly long. They are, after all, written by humans for the purpose of reasoning formally about constraints that arise naturally. Second, it is possible that the predicates that arise in this domain might lend themselves to specialized analyses with good performance. That is, the predicates we actually encounter are not arbitrary n-ary relations over the state space; they are simple tests like "has a database." Third, and most importantly, the constraints that arise are likely to be particularly well-behaved, so the worst-case running time will be quite rare in practice. Of course, this is a claim that is likely to be made about any intractable problem, but at least in our case it can be formalized.

The complexity of the path-checking problem for $\mathsf{LTL}_{\mathsf{AE}}$ arises, of course, from the variable-binding extension, but in particular it arises from *nesting* these binders. All of the intractable examples we have seen, such as the transformed formula in the QBF reduction, involve arbitrarily deeply nested variable bindings. The following theorem shows that model-checking an $\mathsf{LTL}_{\mathsf{AE}}$ path is intractable only to the extent that variables are nested without bound.

Theorem 4. *Let σ be a temporal sequence of length m. Let ϕ be an $\mathsf{LTL}_{\mathsf{AE}}$ formula of length ℓ. Let d be the maximum variable-nesting depth of ϕ (i.e., there is no point in ϕ at which more than d variables are bound at once). Then there is an algorithm that determines whether $\sigma, i \vDash \phi$ in $O(\ell m)$.*

Proof. Let $\phi_1, \ldots, \phi_n$ be the subformulas of ϕ, listed by nondecreasing length. As in theorem 1, we construct a Boolean matrix $[t_{j,k}]_{m \times n}$. We define a recursive algorithm that fills the kth column of the matrix under variable assignment s:

```
FillCol(k, s)
    if φ_k is an atomic proposition
        for j = 1 to m
            t_{j,k} := evaluate φ_k at j using assignment s
    if φ_k = ⊥
        for j = 1 to m
            t_{j,k} := ⊥
```

if ϕ_k is of the form $\phi_g \to \phi_h$
    $\textsc{FillCol}(g, s)$
    $\textsc{FillCol}(h, s)$
    for $j = 1$ **to** m
        $t_{j,k} := t_{j,h}$ **or not** $t_{j,g}$
if ϕ_k is of the form $\bigcirc\phi_h$
    $\textsc{FillCol}(h, s)$
    for $j = 1$ **to** $m - 1$
        $t_{j,k} := t_{j+1,h}$
    $t_{m,k} := \top$
if ϕ_k is of the form $\phi_g \,\mathcal{U}\, \phi_h$
    $\textsc{FillCol}(g, s)$
    $\textsc{FillCol}(h, s)$
    $prev := \bot$
    for $j = m$ **to** 1 **step** -1
        $prev := t_{j,h}$ **or** $(t_{j,g}$ **and** $prev)$
        $t_{j,k} := prev$
if ϕ_k is of the form $\{x\}\phi_h$
    for $j = 1$ **to** m
        $\textsc{FillCol}(h, s[x \to \sigma_j])$
        $t_{j,k} := t_{j,h}$

To solve the model-checking problem, execute $\textsc{FillCol}(n, \emptyset)$ and read the answer from $t_{i,n}$.

This algorithm can be thought of as a sort of hybrid of the algorithms from theorems 1 and 2. Like CHECK, it is recursive and top-down, but it keeps track of intermediate results so they can be reused. Like the algorithm of theorem 1, it uses a matrix to store intermediate results, but in this algorithm results are often overwritten. In evaluating a formula of the form $\{x\}\psi$, we use the columns to our left to first evaluate ψ under the assignment $x \mapsto 1$, then immediately overwrite them, using the same space to evaluate ψ under the assignment $x \mapsto 2$, and so on. In the case where there are no variable assignments, this algorithm reduces to the algorithm of theorem 1; we immediately recurse to the leftmost column, then fill the columns left to right as we travel back up the call stack. In cases with variable assignments, columns are still filled left to right, but sometimes we shift back to the leftmost column and overwrite the old data with results from a new assignment.

The correctness of the algorithm follows easily from the semantics of $\mathsf{LTL}_{\mathsf{AE}}$. I now analyze its complexity. Observe that when this algorithm is

used to check a formula, the kth column of the matrix is ultimately filled m^{d_k} times, where d_k is the variable-nesting depth of ϕ_k (i.e., the number of variables that are bound for the subformula ϕ_k). In particular, no column is filled more than m^d times, where d is the maximum variable-nesting depth of ϕ. Filling a single column, not counting the time to fill its subformula columns to the left, takes $O(m)$ time. And there are n columns in total, which is $O(\ell)$. Therefore the total complexity of the algorithm is $O(\ell m^{d+1})$. □

If d is bounded (e.g., we never have constraints with a variable-nesting depth greater than 3) then we can model-check an $\mathsf{LTL}_{\mathsf{AE}}$ in polynomial time. (If $d = 0$, we get the same, linear performance as the LTL algorithm, as we would hope.) If d is unbounded, then the performance is exponential, $O(\ell m^{\ell})$, since the quantifier depth can approach the length of the formula. In practice, it seems unlikely that variable bindings will often be very deeply nested, since software architects are unlikely to be naturally interested in such convoluted constraints. However, although these theoretical results are quite illuminating, performance testing is needed to understand their practical significance.

2.3 Deductive System

In this section, I will discuss what an axiomatization of $\mathsf{LTL}_{\mathsf{AE}}$ would look like. First, though, I begin with some commentary on the significance of this area. In my project proposal, coming up with a deductive system and proving its soundness and completeness was one of my most significant goals. Since then, I have changed my thinking somewhat and have accordingly de-emphasized this section somewhat, focusing my effort elsewhere.

My original interest in providing a deductive system for $\mathsf{LTL}_{\mathsf{AE}}$ stemmed from three assumptions. First, I thought it would be one of the most theoretically important contributions that my project could offer, providing a novel deductive apparatus for an important extension to LTL. Second, I thought it would be of considerable use for my research. Third, I thought it would be a task for which I was particularly well equipped, given the many axiomatizations and soundness and completeness proofs we have done in class.

All three of these reasons have since been undermined. First, Henzinger [Hen90] has already developed an axiomatization for the related logic TPTL, which I believe could be adapted to $\mathsf{LTL}_{\mathsf{AE}}$ in a fairly straightforward way. Thus, an axiomatization of the variable-binding operator would not be novel or of theoretical importance, unless it somehow improved on Henzinger's

axiomatization. Second, upon thinking more deeply about what results would actually be helpful to my research, I realized that a deductive system would not be particularly useful. The purpose of our constraint language is not to prove general theorems, but instead to check specific evolution models. Thus, it occurred to me that investigating the model-checking problem would be a much more helpful use of my effort, hence the previous section. Third, the axiomatization of the variable-binding operator, and its completeness proof, turn out to be considerably more difficult than most of the deductive systems and proofs we saw in class. Part of the reason for this is that the variable-binding operator is a sort of quantifier, and proving completeness in a quantified logic, such as FOL, requires sophisticated techniques.

For all these reasons, this section does not attempt an exhaustive analysis of the problem of developing a deductive system for $\mathsf{LTL}_{\mathsf{AE}}$, nor do I provide a finished completeness proof. However, I do suggest how we can adapt Henzinger's axiomatization of TPTL to $\mathsf{LTL}_{\mathsf{AE}}$ and how we might be able to prove completeness.

AXIOMATIZATION. We begin by taking LTL as a starting point. There are several alternative axiomatizations for LTL. This one [FHMV95, HMV04] has the advantage of being simple and succinct, and it uses only the $\bigcirc$ and $\mathcal{U}$ operators, just like our syntax.

Definition 6 (deductive system for LTL)**.** We write $\vdash_{\mathsf{LTL}} \phi$ iff $\vdash \phi$ can be derived from the following inference rules and axioms:

Inference rules

$$\frac{\phi \to \psi \quad \phi}{\psi}\ \text{mp} \qquad \frac{\phi}{\bigcirc\phi}\ \text{gen} \qquad \frac{\phi \to \neg\chi \wedge \bigcirc\phi}{\phi \to \neg(\psi\,\mathcal{U}\,\chi)}\ \neg\mathcal{U}$$

Axioms

$$\text{all propositional tautologies} \tag{P}$$
$$\bigcirc\phi \wedge \bigcirc(\phi \to \psi) \to \bigcirc\psi \tag{K}$$
$$\bigcirc\neg\phi \leftrightarrow \neg\bigcirc\phi \tag{$\bigcirc\neg$}$$
$$\phi\,\mathcal{U}\,\psi \leftrightarrow \psi \vee \phi \wedge \bigcirc(\phi\,\mathcal{U}\,\psi) \tag{$\mathcal{U}$}$$

As my labeling of these rules suggests, several of these rules and axioms are already quite familiar. The mp and gen rules are *modus ponens* and the

generalization (or necessitation) rule. The axiom (K) is the Kripke axiom, which tells us that the modality distributes over conditionals. (In class we usually saw (K) in a different form, $\bigcirc(\phi \to \psi) \to (\bigcirc\phi \to \bigcirc\psi)$, but these are equivalent.) Together mp, gen, (P), and (K) are the basic ingredients of a normal modal logic. This tells us that LTL is a normal modal logic with respect to the $\bigcirc$ modality.[3]

Axiom ($\bigcirc\neg$) tells us that $\bigcirc$ is its own dual. Finally, axiom ($\mathcal{U}$) and the $\neg\mathcal{U}$ rule bring $\mathcal{U}$ into the picture; they both express something about the relationship between $\mathcal{U}$ and $\bigcirc$. Axiom ($\mathcal{U}$) is a sort of recursive definition of the $\mathcal{U}$ operator. The $\neg\mathcal{U}$ rule allows us to derive a result of the form $\neg(\psi\,\mathcal{U}\,\chi)$. Taken together, these rules and axioms provide a sound and complete deductive system for LTL [FHMV95].

Unfortunately, this axiomatization breaks down as soon as we allow finite state sequences—before we even get to the interesting changes that $\mathsf{LTL}_{\mathsf{AE}}$ makes to LTL. (Of course, it should. If it *didn't* break down when we changed the semantics, that would tell us that our original axiomatization was unsound or incomplete.) In finite-sequence LTL, which I will call $\mathsf{LTL}_{\mathsf{f}}$, ($\bigcirc\neg$) is no longer valid; it never holds in the final state. In $\mathsf{LTL}_{\mathsf{f}}$, $\bigcirc$ is no longer its own dual; $\bar{\bigcirc}$ is the dual of $\bigcirc$. We will need to weaken ($\bigcirc\neg$) so that it accurately captures the interaction of $\bigcirc$ and $\neg$ in a system that may have only finitely many states.

Axiom ($\mathcal{U}$) is not valid in $\mathsf{LTL}_{\mathsf{AE}}$ either, because the right-hand side of the biconditional can hold even if ψ is never true, for example if we are in the last state, in which case $\bigcirc(\phi\,\mathcal{U}\,\psi)$ is true automatically. To weaken axiom ($\mathcal{U}$) so that it is still valid in finite sequences, we simply need to replace $\bigcirc$ by $\bar{\bigcirc}$ (or alternatively $\mathcal{U}$ by the weak-until operator).

By weakening ($\bigcirc\neg$) and ($\mathcal{U}$) appropriately, we obtain the following axiomatization for LTL over arbitrary sequences (sequences that may be finite or infinite). This axiomatization is due to Pucella [Puc05].

Definition 7 (deductive system for $\mathsf{LTL}_{\mathsf{f}\cup\infty}$)**.** We write $\vdash_{\mathsf{LTL}_{\mathsf{f}\cup\infty}} \phi$ iff $\vdash \phi$ can be derived from the following inference rules and axioms:

Inference rules: mp, gen, $\neg\mathcal{U}$

[3] LTL is also a normal modal logic with respect to the $\Box$ modality; that is, it also satisfies the $\Box$ version of gen and K. So really LTL is a normal bimodal logic, where both $\Box$ and $\bigcirc$ are normal modalities. In fact, an easy way of defining the semantics of the unary fragment of LTL—LTL without $\mathcal{U}$—is to model a temporal Kripke structure as a Kripke structure whose state space is $\mathbb{N}$. Then, as in any multimodal logic, we define an accessibility relation for each modality, which for LTL is simply the successor relation for $\bigcirc$ and $\geq$ for $\Box$.

Axioms: (P), (K), and:

$$\bigcirc\phi \;\leftrightarrow\; (\bigcirc\bot \vee \bar{\bigcirc}\phi) \qquad (\bar{\bigcirc})$$
$$\phi\,\mathcal{U}\,\psi \;\leftrightarrow\; \psi \vee \phi \wedge \bar{\bigcirc}(\phi\,\mathcal{U}\,\psi) \qquad (\bar{\mathcal{U}})$$

To obtain ($\bar{\mathcal{U}}$), we simply changed the $\bigcirc$ in ($\mathcal{U}$) to a $\bar{\bigcirc}$. The new axiom ($\bar{\bigcirc}$), which replaces ($\bigcirc\neg$), relates $\bigcirc$ and $\bar{\bigcirc}$ (but really it is still expressing the relationship between $\bigcirc$ and $\neg$, since $\bar{\bigcirc}$ is merely an abbreviation for $\neg\bigcirc\neg$).

Finally, to obtain an axiomatization for $\mathsf{LTL_f}$, Pucella simply adds an axiom that holds on finite sequences but not infinite ones.

Definition 8 (deductive system for $\mathsf{LTL_f}$)**.**

Inference rules: mp, gen, $\neg\mathcal{U}$

Axioms: (P), (K), ($\bar{\bigcirc}$), ($\bar{\mathcal{U}}$), and:

$$\Diamond\bigcirc\bot \qquad \text{(f)}$$

Now that we have an axiomatization of $\mathsf{LTL_f}$, all that remains is our variable binder, the $\{x\}$ extension. Unfortunately, the axiomatization turns out to be more difficult than the axiom systems we have seen in class. Fortunately, Henzinger [Hen90] did most of the hard work in axiomatizing his half-order modal logic. We can adapt his axiomatization to our setting as follows.

Definition 9 (deductive system for $\mathsf{LTL_{AE}}$)**.**

Inference rules: mp, gen, $\neg\mathcal{U}$, and:

$$\frac{\phi_1 \Rightarrow \cdots \Rightarrow \phi_n \Rightarrow \psi}{\phi_1 \Rightarrow \cdots \Rightarrow \phi_n \Rightarrow \{x\}\psi}\ \{x\}\text{I}$$

where $\phi \Rightarrow \psi$ is an abbreviation for $\phi \rightarrow \Box\psi$ and is right-associative, and where x is not free in $\phi_1, \ldots, \phi_n$.

Axioms: (P), (K), ($\bar{\bigcirc}$), ($\bar{\mathcal{U}}$), (f), and:

$$\{x\}(\phi \rightarrow \psi) \quad\leftrightarrow\quad \{x\}\phi \rightarrow \{x\}\psi \qquad (\{x\}\text{dist})$$
$$\{x\}\phi \leftrightarrow \phi \quad \text{if } x \text{ is not free in } \phi \qquad (\{x\}\text{vac})$$

The new $\{x\}$I rule is a sort of introduction rule (actually a schema that admits infinitely many introduction rules) that allows us to derive formulas with variable bindings. Axiom ($\{x\}$vac) is a sort of elimination axiom that allows us to delete a variable binding only if the bound variable is vacuous—if it does not occur free in the formula. Axiom ($\{x\}$dist) allows us to distribute a variable binding over an implication.

SOUNDNESS AND COMPLETENESS. As usual, proving soundness is easy. The techniques we applied in class suffice. We simply check that each rule and axiom is valid in the semantics. I will not provide the proof here, because it is quite easy.

Completeness is much more challenging, and I do not attempt a proof here. I state completeness as a conjecture and describe how Henzinger's completeness proof for half-order modal logic could be adapted to our system.

Conjecture 1 (completeness of $\mathsf{LTL_{AE}}$)**.** *If* $\vDash_{\mathsf{LTL_{AE}}} \phi$ *then* $\vdash_{\mathsf{LTL_{AE}}} \phi$

Proof Sketch. I believe we can adapt Henzinger's completeness proof for half-order modal logic [Hen90, Hen91]. The structure of Henzinger's proof is typical of a completeness proof for a quantified modal logic; Garson [Gar84] discusses completeness proofs for quantified modal logics in considerable detail. This proof style, in turn, derives from Henkin's well-known proof of the completeness of first-order predicate logic [Hen49].

Before the completeness proof will go through, I think there are a couple of changes we need to make to $\mathsf{LTL_{AE}}$. Recall that in section 2.1 we picked a "flexible" interpretation for function and predicate symbols, meaning that they can take on different values in different states, even if their parameters are identical. For example, in formula (1) the *release* proposition holds in some states but not others. I made this choice because it allows us to express ourselves much more naturally yet does not complicate the semantics. Unfortunately, it *does* complicate the completeness proof. As Garson [Gar84] notes, this style of proof requires rigidity of terms. The reason is that this permits us to reason about equivalence classes of terms under equality, which is a critical step in this style of proof.

Fortunately, in $\mathsf{LTL_{AE}}$ this is easy to do without being disruptive. We can easily convert $\mathsf{LTL_{AE}}$ with flexible symbols into $\mathsf{LTL_{AE}}$ with rigid symbols. Simply change every n-ary symbol into an $(n+1)$-ary symbol, where the new argument specifies the state where the proposition is evaluated. In $\mathsf{LTL_{AE}}$, it is quite clear that this can be done, because we can always introduce variable

binders to refer to the current state. For example, a formula like

$$\Box\{x\}(\mathit{release} \rightarrow \Box \mathit{hasSameFuncAs}(x))$$

becomes

$$\Box\{x\}(\mathit{release}(x) \rightarrow \Box\{y\}\mathit{hasSameFunc}(y, x))$$

Another change we would need to make is to add equality to the language. That is, I based our semantics on first-order logic without equality rather than first-order logic with equality, just to keep our semantics simple. For this style of completeness proof, we need to add equality back in, which would add a new syntax, an extended definition of semantics, and some axioms pertaining to equality.

With all this done, I think Henzinger's proof could be adapted to $\mathsf{LTL}_{\mathsf{AE}}$. The basic idea of the proof is as follows. We begin by introducing a number of new concepts that pertain to sets of formulas: consistency, maximal consistency, completeness, and saturation. A set of formulas Φ is *consistent* iff $\Phi \nvdash \bot$ and *maximally consistent* iff, for every formula ϕ, either $\phi \in \Phi$ or $\neg\phi \in \Phi$. *Completeness* is more complicated; we need to define the $\mathsf{LTL}_{\mathsf{AE}}$ analogue of the FOL notion of ω-completeness. A *saturated* set is one that is complete and maximally consistent. By a process known (in quantified logics generally) as the *Lindenbaum procedure*, we can extend every consistent set that is finite or complete to also be saturated.

Given a consistent formula ϕ, we build a *canonical model*, $M(\phi)$, whose states are certain saturated sets (in particular, the initial state, σ_1 is the saturated extension of $\{\phi\}$). The construction is rather intricate, and I do not provide the details here. The important result we obtain for the canonical model is that for each of its states s (remember that s is a saturated set), $\psi \in s$ iff $M(\phi)[\sigma_1 \mapsto s] \vDash \psi$. (Henzinger calls this the canonical-model theorem. Garson calls it the truth lemma.) We then use this result to show that every consistent formula $\neg\phi$ is satisfiable by its canonical model $M(\neg\phi)$. It follows that $\vdash \phi$ whenever ϕ is a valid formula, which is the desired result. $\Box$

The full proof is quite long and employs a number of sophisticated techniques (some of which, like filtration and unrolling, I have elided in this simplified summary), but it is doable. Since it has already been done by Henzinger, and since I do not anticipate problems adapting the proof specifically to $\mathsf{LTL}_{\mathsf{AE}}$, I have not spent the time or space to write out the proof, instead focusing my efforts elsewhere.

2.4 Expressiveness

I have already referred to the concept of expressiveness a few times. The most basic expressiveness result that we might like to prove is that $\mathsf{LTL}_{\mathsf{AE}}$ is indeed more powerful than LTL. This result is almost trivial, because LTL completely lacks any syntax to express predicates over multiple states, so essentially any $\mathsf{LTL}_{\mathsf{AE}}$ formula involving a polyadic predicate is inexpressible in LTL. But proving the result formally will provide an easy example of the concepts behind expressiveness proofs. Before we can prove anything, we need a working definition of expressive power.

Definition 10 (expressiveness)**.** Let L_1 and L_2 be two linear temporal logics that are defined on the same family of models. We say that a formula ϕ of L_1 is *(initially) expressible* in L_2 if L_2 has a formula ψ such that, on any temporal model (i.e., any temporal sequence σ, any signature, and any interpretation of the signature), $\sigma, 1 \vDash_{L_2} \psi$ holds iff $\sigma, 1 \vDash_{L_1} \psi$ holds.

We say that L_2 is *at least as expressive* as L_1 (written $L_1 \lesssim L_2$) if every formula of L_1 is expressible in L_2. If $L_1 \lesssim L_2$ and $L_2 \lesssim L_1$, then we say L_1 and L_2 are *equally expressive* ($L_1 \sim L_2$). If $L_1 \lesssim L_2$ but not $L_1 \lesssim L_2$, then we say L_2 is strictly *more expressive* than L_1 ($L_1 < L_2$).

There are a few things to note about this definition. There are a number of different definitions of expressiveness, and I have made some important choices in defining it the way I have. First, I have chosen *initial* expressiveness as the kind of expressiveness we are interested in. The alternative would have been a sort of global expressiveness where we require that $\sigma, i \vDash_{L_2} \psi$ holds for all i iff $\sigma, i \vDash_{L_1} \psi$ holds for all i. This distinction does not matter much in most situations, but it does make a difference sometimes. For example, the famous result that past-time operators do not make LTL more expressive does not hold under global expressiveness.

Another choice, in the terminology of Kröger and Merz [KM08], is *model equivalence* versus *logical equivalence*. Model equivalence is what I defined above. Two formulas are logically equivalent if $\phi \leftrightarrow \psi$ holds in the extended logic common to L_1 and L_2. However, this distinction turns out to be moot for the initial sense of expressiveness. For further explanation of these different notions of expressiveness, see section 4.1 of Kröger and Merz.

One final thing to note about this definition is that it is useful only for comparing logics that are defined on the same family of models. This is fine for asking questions like whether a new operator would make LTL more expressive, but it does not allow us to say things like "LTL and the monadic predicate calculus are equally expressive." In fact, this notion of

expressiveness seems to be hard to define formally, without making some appeal to an intuitive notion of "natural" model translations.

Indeed, for us this problem arises immediately, as we have defined $\mathsf{LTL}_{\mathsf{AE}}$ and $\mathsf{LTL}_{\mathsf{f}}$ on different models. Since what we are really interested in is the expressiveness of the variable-binding operator, the best choice here is to compare $\mathsf{LTL}_{\mathsf{AE}}$ with $\mathsf{LTL}_{\mathsf{AE}-\{\}}$, which I define to be $\mathsf{LTL}_{\mathsf{AE}}$ without variable bindings. Note that this ends up looking much like ordinary LTL, since the only predicates we can use are nullary predicates; since $\mathsf{LTL}_{\mathsf{AE}-\{\}}$ lacks variable bindings, there is no way to get a (bound) variable, so there are no (useful) occurrences of non-nullary predicates in $\mathsf{LTL}_{\mathsf{AE}-\{\}}$, except on whatever constants happen to be in a model.

Theorem 5. $\mathit{LTL}_{\mathit{AE}-\{\}} < \mathsf{LTL}_{\mathsf{AE}}$.

Proof. $\mathsf{LTL}_{\mathsf{AE}-\{\}} \lesssim \mathsf{LTL}_{\mathsf{AE}}$ is obvious, since every formula of $\mathsf{LTL}_{\mathsf{AE}-\{\}}$ is also a formula of $\mathsf{LTL}_{\mathsf{AE}}$, with the same semantics. It remains to show $\mathsf{LTL}_{\mathsf{AE}} \not\lesssim \mathsf{LTL}_{\mathsf{AE}-\{\}}$. It suffices to show that the $\mathsf{LTL}_{\mathsf{AE}}$ formula

$$\{x\}p(x)$$

is inexpressible in $\mathsf{LTL}_{\mathsf{AE}-\{\}}$. To prove this, I will construct two different temporal models, which $\{x\}p(x)$ distinguishes but $\mathsf{LTL}_{\mathsf{AE}-\{\}}$ cannot. Let the first model, M, be defined as follows. Let the signature Σ be defined by

$$\Sigma = (\{x\}, (\emptyset), (\emptyset, \{p\}, \emptyset, \emptyset, \ldots))$$

so there is one variable, x, and one unary predicate, p. Now let σ be a sequence of one state, σ_1. Finally, let the interpretation of the predicate be given by $\sigma_1(p) = \emptyset$, so p is never true under any circumstances. Let the second model, N, be defined in the same way, except that we take $\sigma_1(p) = \{\sigma_1\}$, so p is true of σ_1.

Observe that $\sigma, 1 \vDash_{\mathsf{LTL}_{\mathsf{AE}}} \{x\}p(x)$ holds for model N but not for M. However, under the semantics of $\mathsf{LTL}_{\mathsf{AE}-\{\}}$, M and N are indistinguishable (i.e., there is no formula ϕ for which $\sigma, 1 \vDash_{\mathsf{LTL}_{\mathsf{AE}-\{\}}} \phi$ holds for N but not M). To see why this is so, note that the only rule in the semantics of $\mathsf{LTL}_{\mathsf{AE}-\{\}}$ that refers to the interpretation of a predicate at all is the rule that gives the semantics for $p(\pi_1, \ldots, \pi_n)$. Therefore, if there is an $\mathsf{LTL}_{\mathsf{AE}-\{\}}$ formula ϕ that distinguishes between M and N, then p must appear in ϕ. Also, since the predicate symbol p is unary, the occurrence of p in ϕ is syntactically required to have one argument, which must be a term. However, the only term in this model is x, so x must appear in ϕ. However, $\mathsf{LTL}_{\mathsf{AE}-\{\}}$ lacks any syntax by which this variable may be bound, so ϕ cannot be a closed sentence, which contradicts the assumption that $\sigma, 1 \vDash_{\mathsf{LTL}_{\mathsf{AE}-\{\}}} \phi$ holds for N. □

This is not a particularly enlightening proof, but it is a starting point and provides an example for more difficult results. If we wished, we could make this proof more interesting by enriching $\mathsf{LTL}_{\mathsf{AE}-\{\}}$ to give it a "fighting chance" against $\mathsf{LTL}_{\mathsf{AE}}$, for example by adding to $\mathsf{LTL}_{\mathsf{AE}-\{\}}$ terms like now, which would at least allow us to express something like $\{x\}p(x)$. Then we would have to come up with a more interesting $\mathsf{LTL}_{\mathsf{AE}}$ formula to be inexpressible in $\mathsf{LTL}_{\mathsf{AE}-\{\}}$, say $\{x\}\bigcirc p(x)$. But this is straightforward and uninteresting. Suffice it to say that $\mathsf{LTL}_{\mathsf{AE}}$ is more expressive than $\mathsf{LTL}_{\mathsf{AE}-\{\}}$ even if we enrich $\mathsf{LTL}_{\mathsf{AE}-\{\}}$ somewhat; the variable-binding operator genuinely adds expressiveness to the logic.

I now sketch a more interesting result. Due to time constraints, I could not finish the proof fully, but I will at least illustrate how we can use this definition of expressiveness practically. A natural question to ask is whether we can remove operators from $\mathsf{LTL}_{\mathsf{AE}}$ without losing expressiveness. It is well known that the unary fragment of LTL (that is, LTL with $\bigcirc$, $\Box$, and $\Diamond$ but not $\mathcal{U}$) is strictly less expressive than LTL [GPSS80]. But can we remove $\mathcal{U}$ from $\mathsf{LTL}_{\mathsf{AE}}$ without losing expressiveness? Does the expressiveness of the variable-binding operator somehow subsume that of $\mathcal{U}$? This is the case in some logics; for example, $\mathcal{U}$ is definable in certain hybrid logics with $\downarrow$ [BS95], so it is not crazy to wonder if variable bindings might somehow subsume $\mathcal{U}$. But in fact they do not. The following proof sketch is based on a classical proof for LTL [KM08], but proving the result on finite sequences seems to be considerably harder, as I will show. However, adding the variable-binding operator seems to introduce no new difficulties beyond the difficulties of proving the result for $\mathsf{LTL}_{\mathsf{f}}$.

We begin by defining a family of models. We define a signature

$$\Sigma = (\emptyset, (\emptyset), (\{p, q\}, \emptyset, \emptyset, \ldots))$$

so there are no variables, no function symbols, and two nullary predicates, p and q. Now let $\ell, m, n \in \mathbb{N}$. We define a temporal sequence σ with $(n+1)\ell + m + 1$ states: $\sigma_1, \ldots, \sigma_{(n+1)\ell+m+1}$. We define the interpretation of the propositions by

$$\sigma_i(p) = \begin{cases} \bot & \text{if } n+1 \text{ evenly divides } i-m-2 \\ \top & \text{if not} \end{cases}$$

$$\sigma_i(q) = \begin{cases} \top & \text{if } 2n+2 \text{ evenly divides } i-m-2 \\ \bot & \text{if not} \end{cases}$$

We call this model $M(\ell, m, n)$. Define the model $N(\ell, m, n)$ in the same way,

except the interpretation of q is changed to

$$\sigma_i(q) = \begin{cases} \top & \text{if } 2n+2 \text{ evenly divides } i-m-n-3 \\ \bot & \text{if not} \end{cases}$$

The following diagram depicts this setup visually.

$$M(4,m,n) : p,\ldots,p,q,p,\ldots,p,\ \ ,p,\ldots,p,q,p,\ldots,p,\ \ ,p,\ldots,p$$
$$N(4,m,n) : \underbrace{p,\ldots,p}_{m+1},\ \ ,\underbrace{p,\ldots,p}_{n},q,\underbrace{p,\ldots,p}_{n},\ \ ,\underbrace{p,\ldots,p}_{n},q,\underbrace{p,\ldots,p}_{n}$$

That is, each model begins with a string of $m+1$ states where p is true (and q is false). Then, we have a state where the models differ; p is false in both, but q is true in $M(\ell,m,n)$ and false in $N(\ell,m,n)$. Then, we have a string of n states where p is true and q is false. Then, there is another state where the models differ, but this time the assignment is reversed: q is true in $N(\ell,m,n)$ and false in $M(\ell,m,n)$. This pattern is iterated ℓ times. (In the infinite-sequence case, the proof is simpler because ℓ is not needed; we can simply have two infinite models $M(\infty,m,n)$ and $N(\infty,m,n)$ and do not need to reason about entire classes of models.) The goal of the proof is to show that unary $\mathsf{LTL}_{\mathsf{AE}}$, which I will denote $\mathsf{LTL}_{\mathsf{AE}}{-}\mathsf{b}$, cannot distinguish between the M models and the N models, but of course with $\mathcal{U}$ we can.

We now state the following lemma, which leave as a conjecture because I have not been able to flesh out a proof in time for the project deadline:

Conjecture 2. *Let $n \geq m+1$ and let ϕ be an $\mathsf{LTL}_{\mathsf{AE}}{-}\mathsf{b}$ formula with at most m occurrences of $\bigcirc$. Then $M(\ell,m,n),1 \vDash \phi$ holds for all ℓ iff $N(\ell,m,n),1 \vDash \phi$ holds for all ℓ.*

Proof Sketch. Induction on ϕ.

- $\phi = p$. By the defined interpretation, $\sigma_1^{M(\ell,m,n)}(p) = \sigma_1^{N(\ell,m,n)}(p) = \top$. Thus $M(\ell,m,n),1 \vDash p$ and $N(\ell,m,n),1 \vDash p$ for any ℓ.
- $\phi = q$. Similar.
- $\phi = \bot$. Then $M(\ell,m,n),1 \nvDash \phi$ and $N(\ell,m,n),1 \nvDash \phi$ for any ℓ.
- $\phi = \chi \rightarrow \psi$. More difficult than it seems. I may need a more clever induction hypothesis.
- $\phi = \bigcirc\psi$. Then $m \geq 1$. $M(\ell,m,n),1 \vDash \bigcirc\psi$ iff $M(\ell,m,n),2 \vDash \psi$ iff $M(\ell,m-1,n),1 \vDash \psi$ iff $N(\ell,m-1,n),1 \vDash \psi$ iff $N(\ell,m,n),2 \vDash \psi$ iff $N(\ell,m,n),1 \vDash \bigcirc\psi$.

- $\phi = \Box\psi$. Assume $M(\ell, m, n), 1 \not\models \Box\psi$ for some ℓ. Then there is some i with $M(\ell, m, n), i \not\models \psi$. Thus $M(\ell', m', n), 1 \not\models \psi$ for some $\ell' \leq \ell$ and $m' \leq n$. It is easy to show that $N(\ell' + 1, m', n), i + n + 2 \not\models \psi$. Then for any $\ell > \ell' + 1$ we have $N(\ell, m, n), 1 \not\models \Box\psi$. The other direction is similar.

- $\phi = \{x\}\psi$. This case is vacuous because Σ has no variables, so no binding $\{x\}$ is syntactically possible. □

If we can prove this lemma, which essentially says establishes that $\mathsf{LTL}_{\mathsf{AE}}{-}\mathsf{b}$ cannot distinguish between the two families of models, then the desired result follows easily:

Corollary 1. $\mathsf{LTL}_{\mathsf{AE}}{-}\mathsf{b} < \mathsf{LTL}_{\mathsf{AE}}$.

Proof. $\mathsf{LTL}_{\mathsf{AE}}{-}\mathsf{b} \lesssim \mathsf{LTL}_{\mathsf{AE}}$ is obvious. We must show $\mathsf{LTL}_{\mathsf{AE}} \not\lesssim \mathsf{LTL}_{\mathsf{AE}}{-}\mathsf{b}$. It suffices to show that $p\,\mathcal{U}\,q$ is inexpressible in $\mathsf{LTL}_{\mathsf{AE}}{-}\mathsf{b}$. Assume for contradiction that $p\,\mathcal{U}\,q$ is expressible by some $\mathsf{LTL}_{\mathsf{AE}}{-}\mathsf{b}$ formula ϕ, and let m denote the number of times that $\bigcirc$ occurs in ϕ.

Observe that $M(\ell, m, m{+}1), 1 \vDash p\mathcal{U}q$ and $N(\ell, m, m{+}1), 1 \not\models p\mathcal{U}q$ for any ℓ. By assumption we then have $M(\ell, m, m{+}1), 1 \vDash \phi$ and $N(\ell, m, m{+}1), 1 \not\models \phi$ for all ℓ. This contradicts conjecture 2. □

We could do more comparisons along the same lines. It is worth noting that expressiveness proofs can be somewhat difficult in general, and even apparently simple statements can have complicated proofs; as I have just shown, even rather pedestrian results can have complications. Logicians have developed a few tools for proving expressiveness results: bisimulations, generated substructures, and others. See Blackburn and Seligman [BS95] for a discussion in the context of hybrid languages.

Another result I would like is a description of the expressiveness of $\mathsf{LTL}_{\mathsf{AE}}$ in terms of first-order logic. It is well known that LTL is equivalent in expressive power to the monadic predicate calculus over $(\mathbb{N}, <)$ [Kam68]. There is a natural and elegant embedding of $\mathsf{LTL}_{\mathsf{AE}}$ in the full (i.e., polyadic) first-order predicate calculus over $(\mathbb{N}, <)$, which I will presently illustrate. Much as we did in section 2.3 when sketching a completeness proof for a deductive system, we are going to make a slight change to $\mathsf{LTL}_{\mathsf{AE}}$ that will make the translation much easier and more natural: we give all function and predicate symbols a *rigid* interpretation, so that they are not state-dependent. As I showed in section 2.3, this is permissible because we can easily translate a flexibly interpreted instance of $\mathsf{LTL}_{\mathsf{AE}}$ into a rigidly interpreted instance.

Now, let $\Sigma = (V, (F_n)_n, (P_n)_n)$ be an $\mathsf{LTL_{AE}}$ signature. We define a function F_i that maps $\mathsf{LTL_{AE}}(\Sigma)$ formulas to $\mathsf{FOL}(\Sigma)$ formulas as follows.

$$\begin{aligned} F_i(p(\pi_1, \ldots, \pi_n)) &= p(\pi_1, \ldots, \pi_n) \\ F_i(\bot) &= \bot \\ F_i(\phi \to \psi) &= F_i(\phi) \to F_i(\psi) \\ F_i(\bigcirc \phi) &= F_{i+1}(\phi) \\ F_i(\phi \,\mathcal{U}\, \psi) &= \exists j (j \geq i \wedge F_j(\psi) \wedge \forall i (i \leq k < j \to F_k(\phi))) \\ F_i(\{x\}) &= F_i(\phi)[i/x] \end{aligned}$$

To translate a closed $\mathsf{LTL_{AE}}(\Sigma)$ sentence ϕ, we invoke $F_1(\phi)$. We interpret the domain of quantification as $\mathbb{N}$ and translate the interpretations of the function and predicate symbols in the obvious way, mapping a state σ_i to the natural number i. A constraint like

$$\Box\{x\}(\mathit{release}(x) \to \Box\{y\}\mathit{hasAllFunc}(y, x))$$

translates to a first-order formula like

$$\forall x(\mathit{release}(x) \to \forall y(y \geq x \to \mathit{hasAllFunc}(y, x)))$$

This translation is natural and elegant. But is $\mathsf{LTL_{AE}}$ *completely* expressive with respect to first-order logic, in the same way that any formula of *monadic* predicate logic can be translated back into LTL? Or if not, what fragment of first-order logic does $\mathsf{LTL_{AE}}$ cover? I do not yet have the answers to these questions, but I think they are important future work.

3 Conclusion

In this section, I begin with a somewhat informal reflection on the surprises this project has dealt and what I have learned, then enumerate some of the results I have obtained and their significance, and finally suggest future work.

REFLECTION. Over the course of this project, I have confronted a number of surprises, and while I did much of what I aimed to do and got out of the project what I wanted, my focus changed somewhat.

The first surprise was the tremendous volume of related work. Two months ago, despite having in the past attempted a couple of literature

searches on this topic, I was aware of only one or two papers that included something like the variable-binding syntax I had invented, and these papers did not provide any theoretical insight into the operator. I found the lack of papers surprising, since state capture seemed an obvious extension to make to LTL, but before I began this project my efforts to find related work had met with limited success.

In fact, as I have documented earlier in this report, there is a rich literature on this kind of extension to LTL, and many important theoretical results have been obtained. I attribute my ignorance of them to the somewhat fragmented character of the literature on this topic. The plethora of names alone makes conducting a conventional literature search difficult. Off the top of my head, I count at least eight names for this variable-binding idea that I am aware of:

- *Temporal quantification*, Alur and Henzinger's original name for this idea in the context of real-time systems [AH89]
- *Freeze quantification*, Alur and Henzinger's revised name (reportedly suggested to them by Amir Pnueli himself) [Hen90]
- *Half-order modal logic*, Henzinger's generalization of the concept to modal logic [Hen90]
- *Rigid variables* [Ric92]—this was the paper I became aware of first and was therefore the name we used in our paper [GBSC09]
- *Reference pointers* [Gor94]
- $\downarrow$ *binders* (and more generally *nominals*), the terminology used in hybrid logic [BT99]
- *Registers* [DL06]
- *Memoryful logic* [DLS10]

Indeed, the idea seems to have been independently invented multiple times; the authors of some of these papers are evidently as unaware of the related work as I was when I developed the idea for our path constraint language. The main line of work seems to be that initiated by Alur and Henzinger; *freeze quantification* seems to be the name by which the idea is most popularly recognized. (Deservedly so; while it is possible that the idea occurred to others before Alur and Henzinger, they were certainly the first to grapple with the theoretical issues in any serious way. Henzinger's doctoral thesis

[Hen91] details the impressive array of theoretical results that they were able to achieve.)

This fragmented literature seems to be a consequence of the way it arose. As in our case, researchers have usually invented the concept to solve particular problems in their respective research areas: real-time systems, data models, artificial intelligence, and now software architecture. In any case, I could almost say that coming to terms with this literature was one of the most important outcomes of this project in terms of direct benefit to my research—more so than any of the results I derived.

I encountered a number of surprises in attempting to carry out the tasks I had set for myself. One surprise was that some of my goals had already been accomplished by other researchers for related logics. For example, Henzinger's axiomatization of PLTL translates rather directly into an axiomatization for $\mathsf{LTL}_{\mathsf{AE}}$. This caused me to alter the focus of my project in some ways and also to attempt some new results that I had not previously planned to work on.

In particular I am referring to the section on model checking (section 2.2); model checking was not something I planned to work on at all for this project. I considered the infinite-path, arbitrary-state-structure problem irrelevant (and already solved by Alur and Henzinger [AH89]) and assumed that the single-finite-path problem would be trivial, which turned out to be very much not the case. My results on model checking, I think, culminating in theorem 3, were one of the most successful and difficult outcomes of this project.

A number of problems, like this model-checking problem, turned out to be considerably harder than I anticipated. Ultimately my project took much more time than I had planned, occupied many more pages than I expected, and required much deeper and broader knowledge than I possessed at the outset of this project. This gave me an opportunity to immerse myself in several fields of which I previously had very limited knowledge: model checking, complexity theory, and others.

On the whole, I am quite pleased with the outcome of my project. Although some of my initial goals turned out to be naive in some respects, I exposed myself to a tremendous amount of relevant information and obtained some important results. In addition, I have succeeded in my main goal, which was to ensure that our path constraint language is on firm theoretical footing and to clarify its position within a broader family of related logics that have been devised for other domains.

SUMMARY OF RESULTS. I will now present a summary of the main results of this paper. For each result, I will describe both its relevance to my research and its broader interest as a result in logic.

- I *surveyed the literature* of related logics and compared them with $\mathsf{LTL}_{\mathsf{AE}}$.

  Impact on my research: Coming to grips with the broad literature of related logics was, as I have argued above, one of the most important outcomes of this project. A few months ago, I was quite ignorant of the great majority of this literature. Being able to benefit from two decades of existing theoretical work will be enormously helpful.

  Broader significance: As I opined above, the literature on this topic is somewhat fragmented, so I think examining it in a careful, inclusive way is a very useful endeavor. In this report I do not purport to offer a comprehensive, encyclopedic literature review. But I do think I have been reasonably successful in characterizing the extent of this variable-binding idea and how it has developed over time, and perhaps pointing to a need for a more unified understanding.

- I developed *formal descriptions* of the logic: a syntax and a Kripke-style semantics.

  Impact on my research: This exercise forced me to think rigorously about the underpinnings of our LTL extension. There were a number of questions about the syntax and semantics on which I was fuzzy. Where should variable bindings be allowed? How can we model user-defined predicates? Simply writing down a formal syntax and semantics was very useful for clarifying these points. Of course, these results will also be of practical use, for example in implementing $\mathsf{LTL}_{\mathsf{AE}}$ in tools.

  Broader significance: A number of formal descriptions of similar operators have been developed previously. I think $\mathsf{LTL}_{\mathsf{AE}}$ occupies an interesting position within this field of related logics, and a few interesting differences emerged in writing down the formal descriptions.

- I considered the *model-checking* problem for (single) $\mathsf{LTL}_{\mathsf{AE}}$ paths and proved the problem to be PSPACE-complete (or polynomial-time if the variable-nesting depth has a hard upper bound).

Impact on my research: As I have explained, the path-checking problem is of considerable, immediate importance for my research. Understanding the complexity of the problem is very helpful. I am quite pleased that I was able to obtain such precise results here; I learned a lot of complexity theory to finish this part of the project, and I think it was time well spent. What the results tell us is that path checking is computationally difficult in the worst case, but not too bad as long as binding operators are not deeply nested.

Broader significance: In terms of novel research, I think the section on model checking turned out to be one of the most important and successful sections of this project report. I used a novel reduction of QBF to prove an important result about the complexity of model-checking a finite $\mathsf{LTL}_{\mathsf{AE}}$ path—a result that seems to be generalizable to other logics with similar variable-binding operators.[4]

- I suggested how we could develop a *deductive system* for $\mathsf{LTL}_{\mathsf{AE}}$ and described what a completeness proof would look like.

  Impact on my research: As I have explained at length, I do not think a deductive system is likely to be particularly helpful for my research, which is one reason that I focused my efforts elsewhere for this project. However, it may be useful for manual reasoning about theorems of $\mathsf{LTL}_{\mathsf{AE}}$.

  Broader significance: The broader significance of this work is limited, since it is little more than adaptation of previous work. The main respect in which it differs from Henzinger's axiomatization of half-order logic is that is specialized to finite state sequences.

- I introduced a framework for reasoning about the *expressiveness* of $\mathsf{LTL}_{\mathsf{AE}}$ and illustrated a couple of proofs.

[4] Alas, my QBF transformation, and the result it proves, turns out not to be a bit less novel than I thought. On the day of the project deadline, I discovered that Demri et al. [DLS10], in a paper to be published in the May 17, 2010, issue of *Theoretical Computer Science*, use a similar QBF transformation to prove PSPACE-completeness for the finitary model-checking problem for LTL augmented with finitely many registers, over deterministic one-counter automata. I must admit that I do not understand the details of their domain, but their reduction technique bears a strong resemblance to mine, and their result, while somewhat restricted, is closely related. This is the first time I have been scooped from ten days in the future, but at least this bolsters my confidence that my proof is correct!

Impact on my research: The inadequate expressiveness of LTL was the original motivation for extending it, so the expressiveness of $\mathsf{LTL}_{\mathsf{AE}}$ is very important to us. This project succeeded in validating my intuition that $\mathsf{LTL}_{\mathsf{AE}}$ is substantially more expressive than LTL. It also illustrated a natural embedding of $\mathsf{LTL}_{\mathsf{AE}}$ within first-order predicate logic, although some interesting questions about this embedding are left for future work, including notably whether the embedding is complete (and, if not, precisely what fragment of FOL it occupies).

Broader significance: The expressiveness of variable-binding operators of various kinds has been considered in the literature previously, although often in a somewhat haphazard and domain-specific manner. The situation for hybrid logic is fairly well understood, thanks largely to Blackburn and Seligman [BS95] and others thinking rigorously about expressiveness in the 1990s. The situation for TPTL is also well understood, but I am not sure that the results from TPTL generalize readily beyond that domain. In general, there seem to be a number of competing definitions and understandings of expressiveness; it is a more difficult concept to define than it at first appears. My results here are broadly in line with previous work; none of my results were particularly surprising. What was somewhat surprising was that proving expressiveness results in a finite-sequence setting seems to be actually *more* difficult than proving them for infinite paths. For example, proving that binary $\mathsf{LTL}_{\mathsf{f}}$ is more expressive than unary $\mathsf{LTL}_{\mathsf{f}}$ ought to be easy, but in fact both the model construction and the proof itself were considerable more challenging than I expected. My understanding of the more sophisticated techniques for reasoning about expressiveness is limited, but it seems to me that reasoning about expressiveness on finite paths might be an interesting thing to investigate.

FUTURE WORK. This project report leaves a number of questions unanswered. Indeed, there were several would-be theorems that ended up as conjectures. I do not think that any of these questions are impenetrably difficult; probably I could work out each of them with a few more days of effort.

These conjectures do vary somewhat in character. Conjecture 1, the completeness conjecture, is likely to be a reasonably straightforward but

very tedious adaptation of Henzinger's completeness proof for half-order logic. In addition, it is unlikely to be of much importance to my research, to which deductive systems are not of particular relevance, nor to logic research in general, since Henzinger has already presented a solution to the problem. On the other hand, the questions at the end of section 2.4, I think, are important and not adequately answered by the existing literature.

I think there is a lot of interesting work beyond the scope of this project too. For example, this project approaches the problem from the standpoint of evaluating a constraint language that we have developed and comparing it to other logics that are similar. An alternate approach would be to explore how a variety of formal systems could be adapted to express architecture evolution constraints, and consider their relative merits. For example, what would happen if we developed a constraint language based on hybrid logic rather than LTL? Would the additional constructs that hybrid logic provides, such as satisfaction statements, be useful in expressing evolution constraints? Or what about a branching-time logic? In our model of architecture evolution, it is natural to model the set of prospective evolution paths as a tree, so a branching-time logic superficially seems like a natural fit.

There are also a number of natural future directions that pertain specifically to my research. For example, in addition to this path constraint language, there are a number of other formal languages that will be involved in specifying an architecture evolution. There is the architecture description language itself, such as Acme. I am also working on a way to specify individual architectural transformations, which model discrete steps within an architecture evolution path. These modeling languages interact in interesting ways. For example, individual architectural transformations are likely to be somewhat limited with respect to how much they can change an architectural model in a single step. That is, step 11 in an evolution path is likely to look a lot like step 10, with only incremental changes. There is a natural regularity and steadiness to a planned architecture evolution. Can we exploit these properties somehow to, say, improve the performance of model checking?

Although this project is not the final word, not even the final word on $\mathsf{LTL}_{\mathsf{AE}}$, it lays an excellent foundation for asking these kinds of questions. The results that I have obtained in this project have equipped me with a broad perspective that will be very helpful as I continue my research in software architecture evolution.

4 References

[AH89] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. In *Proc. Symposium on Foundations of Computer Science (FOCS'89)*, pages 164–169. IEEE, 1989.

[AH94] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1):181–204, 1994.

[Bla00] Patrick Blackburn. Internalizing labelled deduction. *Journal of Logic and Computation*, 10(1):137–168, 2000.

[BLS07] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. Technical Report TUM-I0724, Technische Universität München, 2007. To appear in *ACM Transactions on Software and Methodology*.

[BS95] Patrick Blackburn and Jerry Seligman. Hybrid languages. *Journal of Logic, Language, and Information*, 4(3):251–272, 1995.

[BT99] Patrick Blackburn and Miroslava Tzakova. Hybrid languages and temporal logic. *Logic Journal of the IGPL*, 7(1):27–54, 1999.

[Cle96] Paul C. Clements. A survey of architecture description languages. In *Proc. International Workshop on Software Specification and Design (IWSSD'96)*, pages 16–25. IEEE, 1996.

[DL06] Stéphane Demri and Ranko Lazić. LTL with the freeze quantifier and register automata. In *Proc. Symposium on Logic in Computer Science (LICS'06)*, pages 17–26. IEEE, 2006.

[DLS10] Stéphane Demri, Ranko Lazić, and Arnaud Sangnier. Model checking memoryful linear-time logics over one-counter automata. *Theoretical Computer Science*, 411:2298–2316, 2010.

[FdR06] Massimo Franceschet and Maarten de Rijke. Model checking hybrid logics (with an application to semistructured data). *Journal of Applied Logic*, 4(3):279–304, 2006.

[FHMV95] Ronald Fagin, Joseph Y. Halperin, Yoram Moses, and Moshe Y. Vardi. *Reasoning about Knowledge*, section 8.1. MIT Press, 1995.

[Gar84] James W. Garson. Quantification in modal logic. In *Extensions of Classical Logic*, volume 2 of *Handbook of Philosophical Logic*, pages 249–307. Kluwer, 1984.

[GBSC09] David Garlan, Jeffrey M. Barnes, Bradley Schmerl, and Orieta Celiku. Evolution styles: Foundations and tool support for software architecture evolution. In *Proc. Working IEEE/IFIP Conference on Software Architecture (WICSA'09)*, pages 131–140. IEEE, 2009.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability*. Freeman, 1979.

[GMW00] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In *Foundations of Component-Based Systems*, pages 47–67. Cambridge Univ. Press, 2000.

[Gor94] Valentin Goranko. Temporal logic with reference pointers. In *Proc. International Conference on Temporal Logic (ICTL'94)*, pages 133–148. Springer, 1994.

[GPSS80] Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *Proc. Symposium on Principles of Programming Languages (POPL'80)*, pages 163–173. ACM, 1980.

[Gru05] Lars Grunske. Formalizing architectural refactorings as graph transformation systems. In *Proc. International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'05)*, pages 324–329. IEEE, 2005.

[Hen49] Leon Henkin. The completeness of the first-order functional calculus. *Journal of Symbolic Logic*, 14(3):159–166, 1949.

[Hen90] Thomas A. Henzinger. Half-order modal logic: How to prove real-time properties. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 281–296. ACM, 1990.

[Hen91] Thomas A. Henzinger. *The Temporal Specification and Verification of Real-Time Systems*. PhD thesis, Stanford Univ., 1991.

[HKN+07] Christine Hofmeister, Philippe Kruchten, Robert L. Nord, Henk Obbink, Alexander Ran, and Pierre America. A general model of software architecture design derived from five industrial approaches. *Journal of Systems and Software*, 80(1):106–126, 2007.

[HMV04] Joseph Y. Halperin, Ron van der Meyden, and Moshe Y. Vardi. Complete axiomatizations for reasoning about knowledge and time. *SIAM Journal on Computing*, 33(3):674–703, 2004.

[Kam68] Johan Anthony Willem Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, Univ. California, Los Angeles, 1968.

[KM08] Fred Kröger and Stephan Merz. *Temporal Logic and State Systems*. Springer, 2008.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[MS03] N. Markey and P. Schnoebelen. Model checking a path. In *Proc. International Conference on Concurrency Theory (CONCUR'03)*, volume 2761 of *Lecture Notes in Computer Science*, pages 251–265. Springer, 2003.

[Puc05] Riccardo Pucella. The finite and the infinite in temporal logic. *ACM SIGACT News*, 36(1):86–99, 2005.

[Ric92] Joel Richardson. Supporting lists in a data model (a timely approach). In *Proc. International Conference on Very Large Data Bases (VLDB'92)*, pages 127–138. Morgan Kaufmann, 1992.

[SC85] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

[SG01] Bridget Spitznagel and David Garlan. A compositional approach for constructing connectors. In *Proc. Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, pages 148–157. IEEE, 2001.

[SOV02] Christoph Stoermer, Liam O'Brien, and Chris Verhoef. Practice patterns for architecture reconstruction. In *Proc. Working Conference on Reverse Engineering (WCRE'02)*, pages 151–160. IEEE, 2002.

[WF02] Michel Wermelinger and José Luiz Fiadeiro. A graph transformation approach to software architecture reconfiguration. *Science of Computer Programming*, 44(2):133–155, 2002.