CS 295A/395D: Artificial Intelligence

LTL Applications

Prof. Emma Tosch

18 April 2022

The University of Vermont

Logistics

Reminders:

- Worksheet due Friday in class
    - Remember: option to peer grade; must be in class to swap
- Programming assignment out
- When running into issues, please post questions on Teams!

Agenda

LTL

- Review syntax and semantics
- Example modeling problem
- Model Checking
- Applications of LTL

Recall: LTL Syntax

LTL Syntax:

$$\phi ::= p \mid \top \mid \bot \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \rightarrow \phi_2$$

$$\mid X\phi \mid G\phi \mid F\phi \mid \phi_1 U \phi_2 \mid \phi_1 W \phi_2 \mid \phi_1 R\, \phi_2$$

Recall: LTL is propositional logic + X + at least one of {U, W, R}

(We will typically just use {X, U})

Recall: LTL Semantics

LTL Semantics:

$$\mathcal{M} ::= < S, R, L >$$

Labelled finite automata:

- S = states
- R = edges
- L = labelling function

Recall: must be able to produce infinite paths (no deadlock!)

Recall: LTL Semantics

LTL Semantics:

$\pi \vDash X\phi \leftrightarrow \pi^2 \vDash \phi$

$\pi \vDash G\phi \leftrightarrow \forall i\left(\pi^i \vDash \phi\right)$

$\pi \vDash F\phi \leftrightarrow \exists i\left(\pi^i \vDash \phi\right)$

$\pi \vDash \phi_1 U\, \phi_2 \leftrightarrow \exists i(\pi^i \vDash \phi_2 \land \forall j(j < i \rightarrow \pi_i \vDash \phi_1))$

$\pi \vDash \phi_1 W\, \phi_2 \leftrightarrow \pi \vDash G\phi_1 \lor (\pi \vDash \phi_1 U\, \phi_2)$

$\phi \vDash \phi_1 R\, \phi_2 \leftrightarrow \pi \vDash G\phi_2 \lor \exists i\, (\pi^i \vDash \phi_1 \land \forall\left(j \leq i \rightarrow \pi^j \vDash \phi_2\right))$

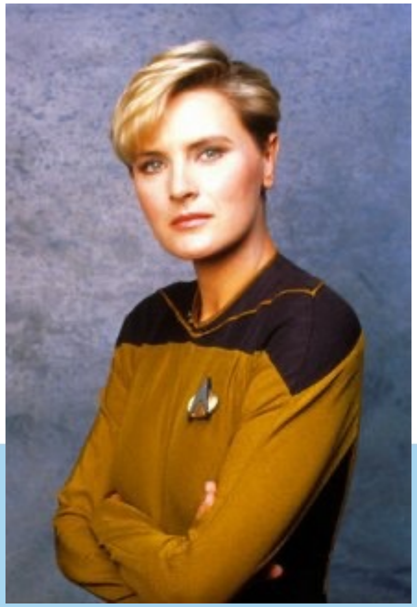
LTL Exercises

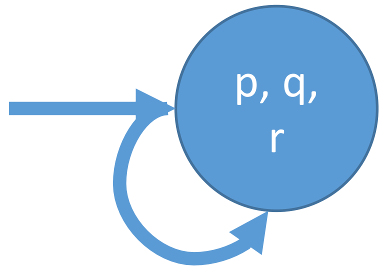
Example:

p = Yar alive

q = Yar on our ship

r = transporter ready





Scenario: Hit by enemy before transport

State: Yar dead, not on ship

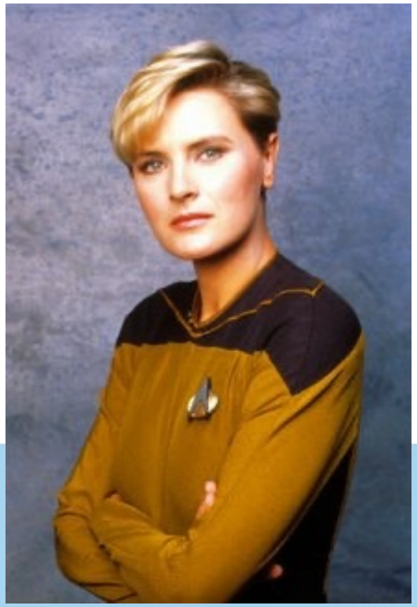
LTL Exercises

Example:

p = Yar alive

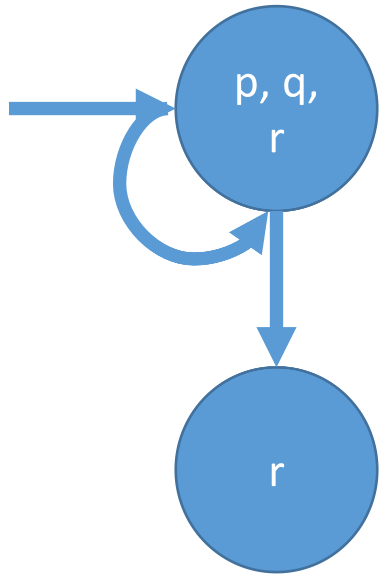
q = Yar on our ship

r = transporter ready



Scenario: Hit by enemy before transport

State: Yar dead, not on ship


$$\begin{tikzcd} \arrow[r] & p, q, r \arrow[loop left] \arrow[d] \\ & r \end{tikzcd}$$

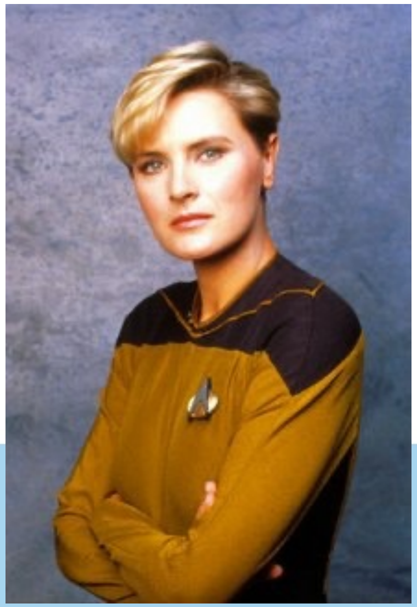
LTL Exercises

Example:

p = Yar alive

q = Yar on our ship

r = transporter ready



Scenario: Hit by enemy before transport

State: Yar dead, not on ship

Transporter could be up or down (hit or not hit)

$$\begin{tikzcd}
\arrow[r] & p, q, r \arrow[loop left] \arrow[dl, leftrightarrow] \arrow[dr] & \\
r & &
\end{tikzcd}$$

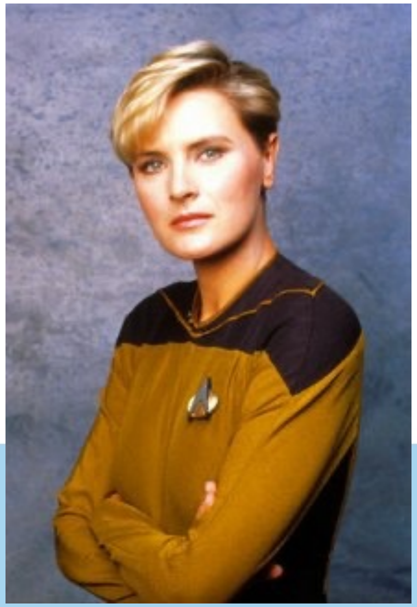
LTL Exercises

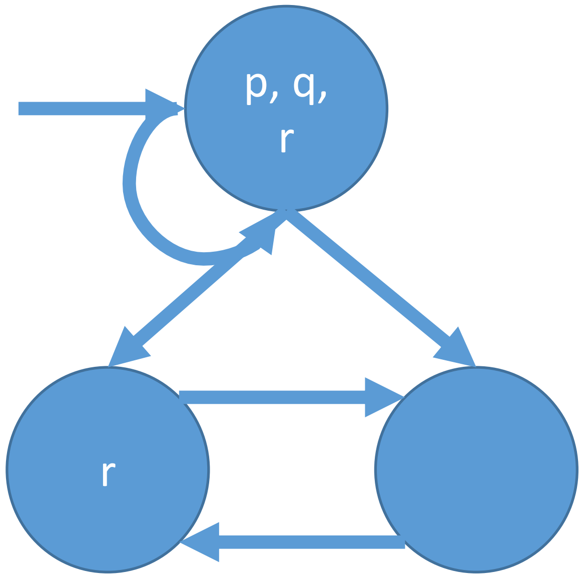
Example:

p = Yar alive

q = Yar on our ship

r = transporter ready





Scenario: Hit by enemy before transport

State: Yar dead, not on ship

Transporter could be up or down (hit or not hit)

Transporter can cycle

It takes time to charge

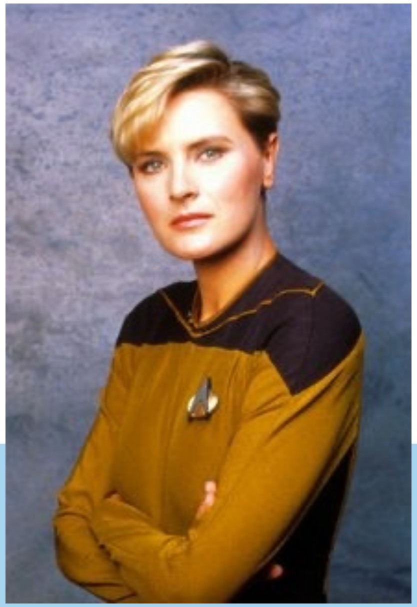
LTL Exercises

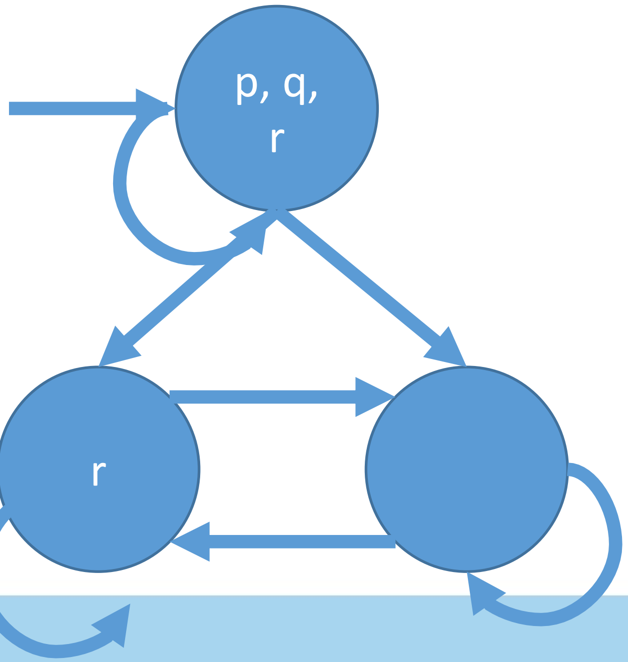
Example:

p = Yar alive

q = Yar on our ship

r = transporter ready





Scenario: Hit by enemy before transport

State: Yar dead, not on ship

Transporter could be up or down (hit or not hit)

Transporter can cycle

It takes time to charge

LTL Exercises

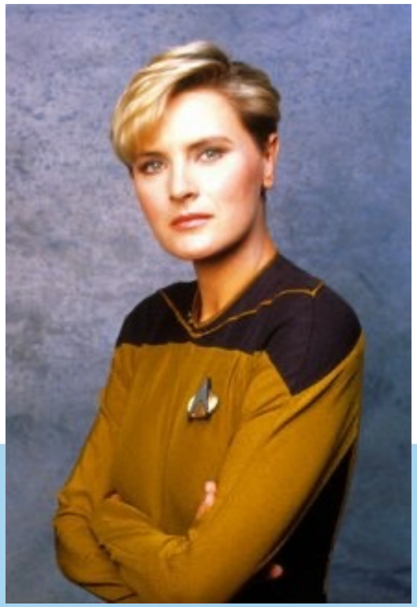
Scenario: Yar transports onto enemy ship

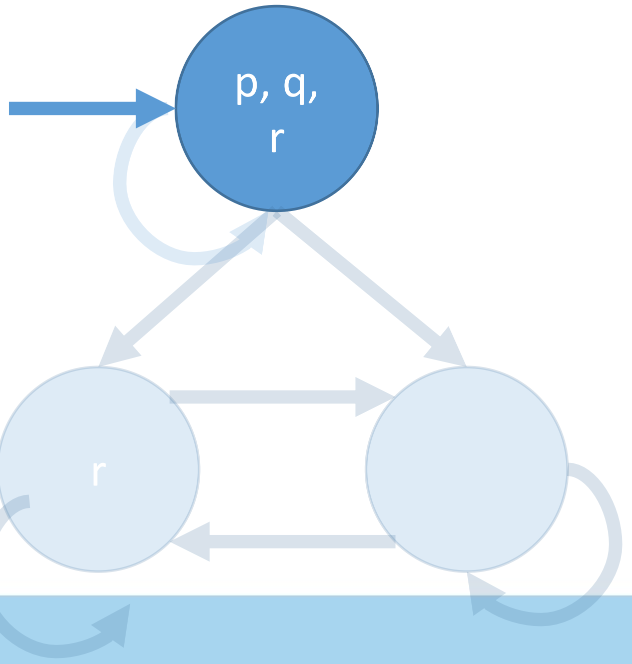
Example:

p = Yar alive

q = Yar on our ship

r = transporter ready





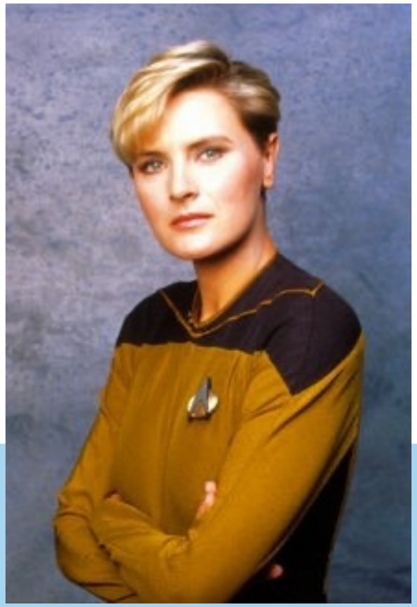
LTL Exercises

Example:

p = Yar alive

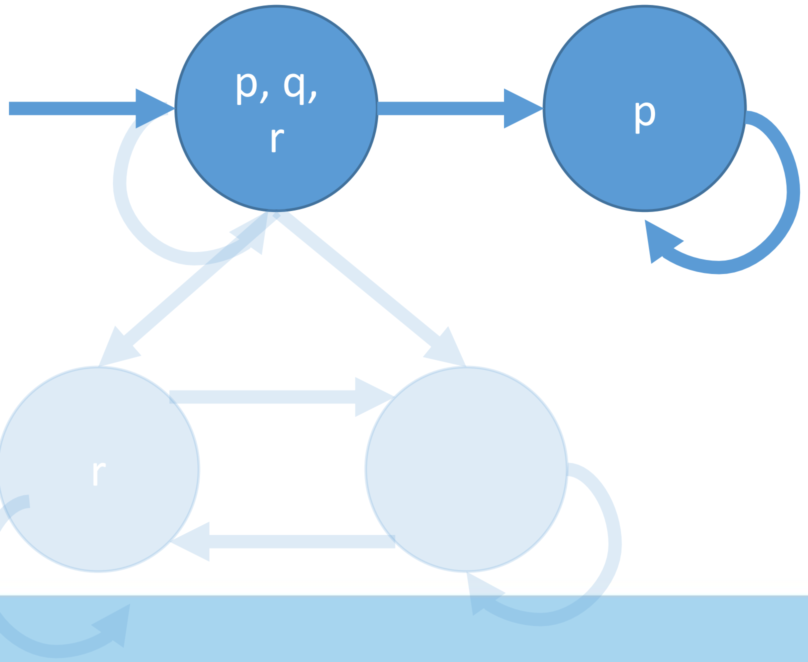
q = Yar on our ship

r = transporter ready



Scenario: Yar transports onto enemy ship

Transporter takes time to recharge



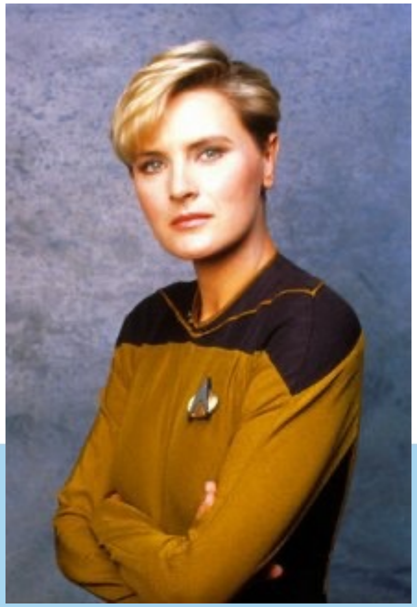
LTL Exercises

Example:

p = Yar alive

q = Yar on enemy ship

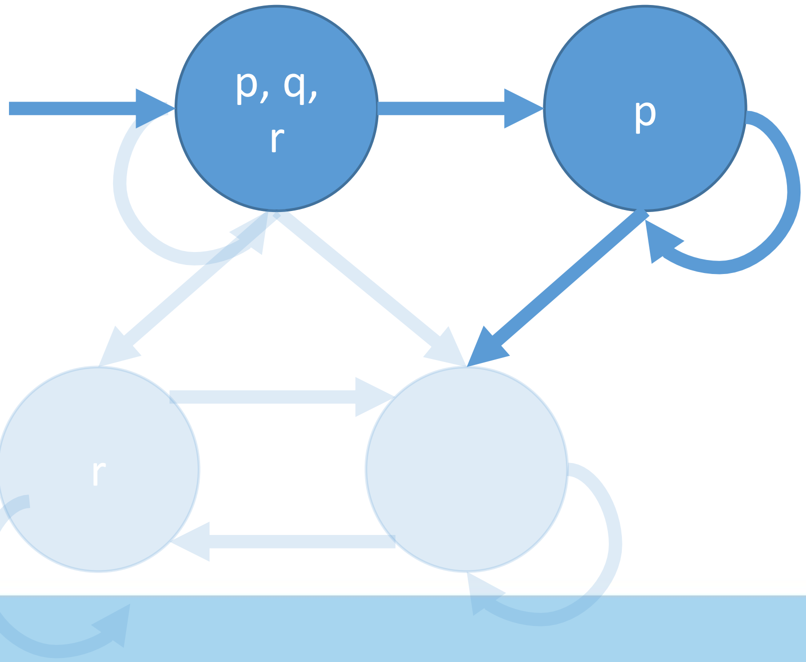
r = transporter ready



Scenario: Yar transports onto enemy ship

Transporter takes time to recharge

Yar could die in this time



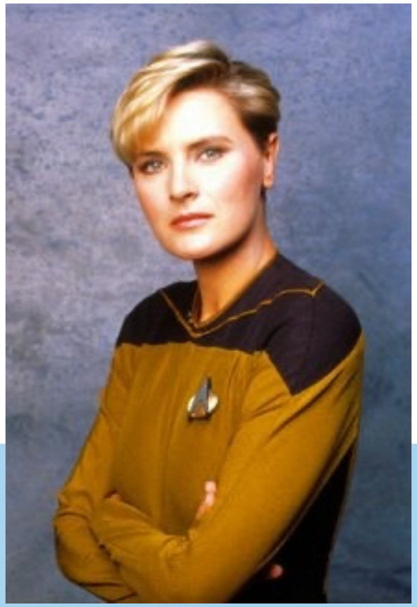
LTL Exercises

Example:

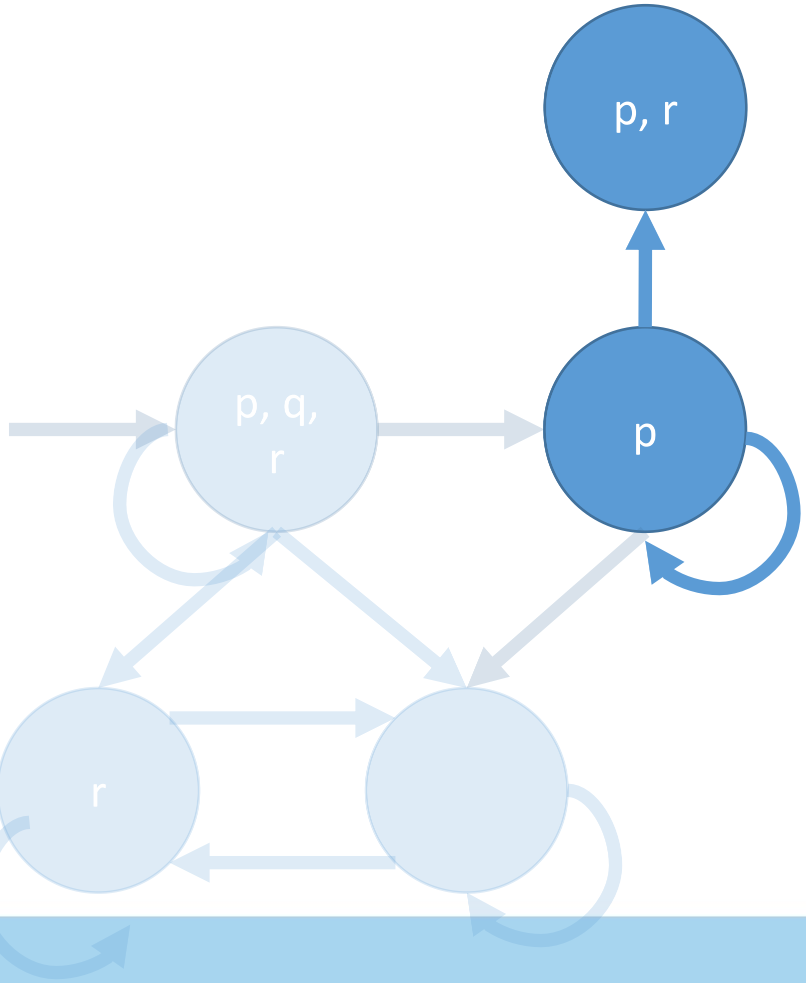
p = Yar alive

q = Yar on our ship

r = transporter ready



Scenario: Yar transports onto enemy ship



Transporter becomes ready

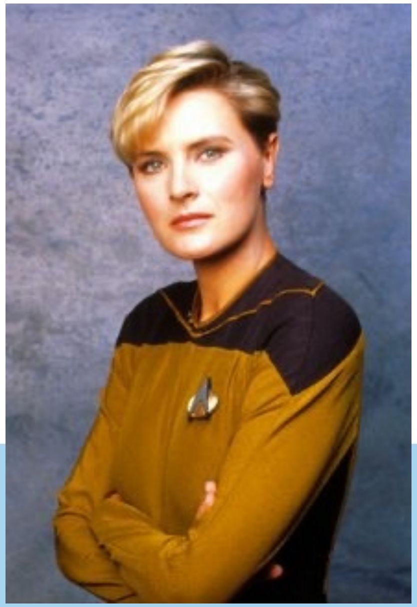
LTL Exercises

Example:

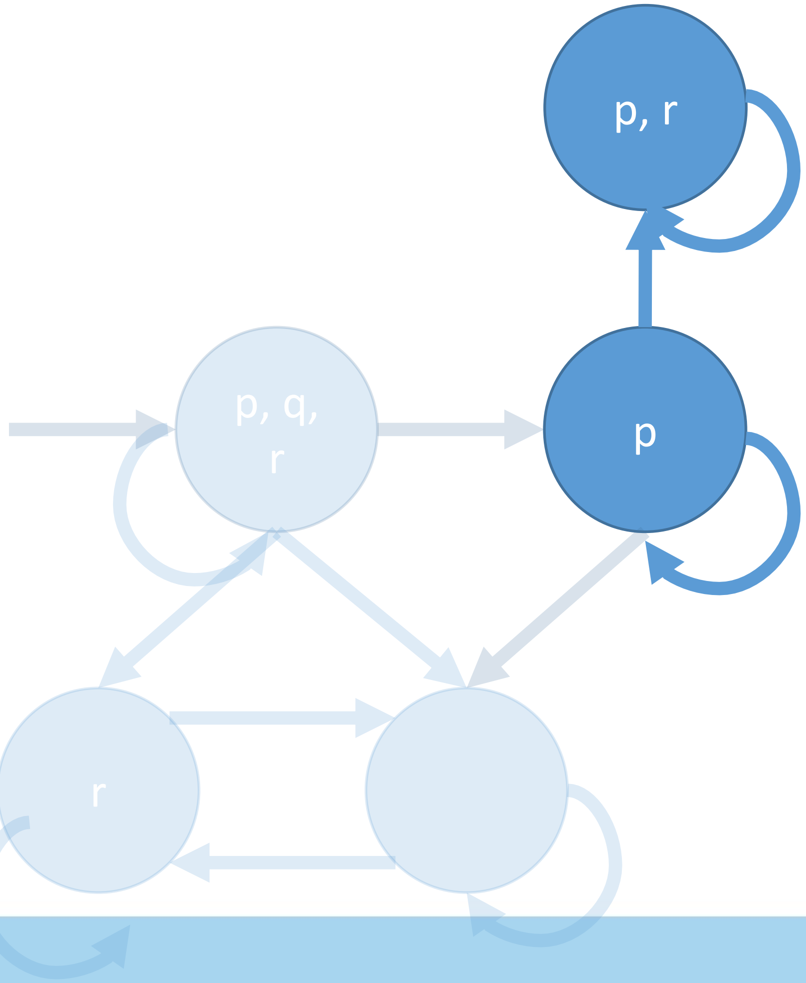
p = Yar alive

q = Yar on our ship

r = transporter ready



Scenario: Yar transports onto enemy ship



Transporter becomes ready

Yar has health; fights some more

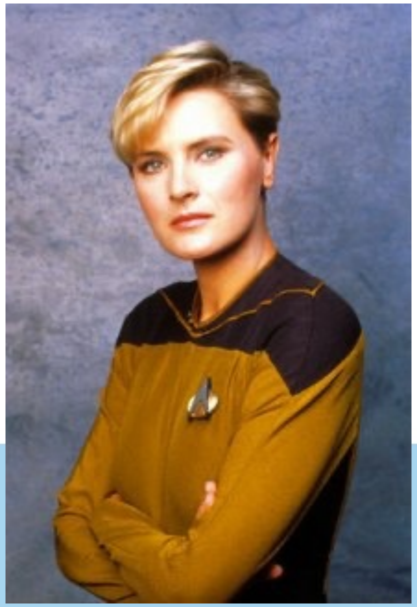
LTL Exercises

Example:

p = Yar alive

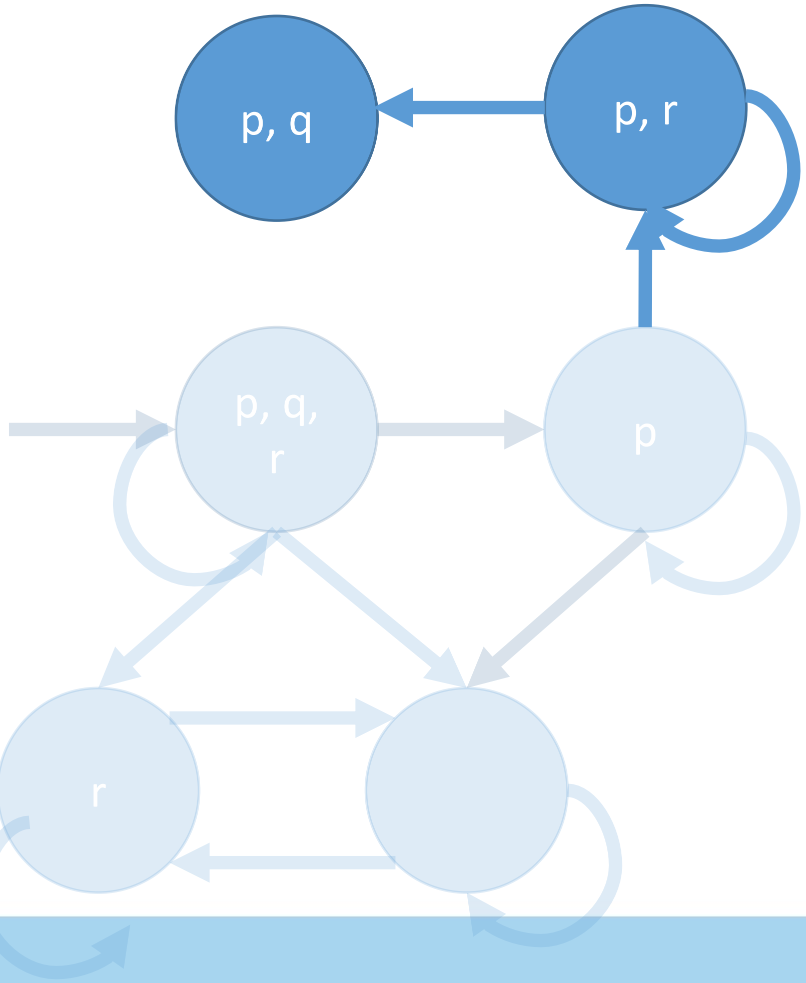
q = Yar on our ship

r = transporter ready



Scenario: Yar transports onto enemy ship

Yar transports back



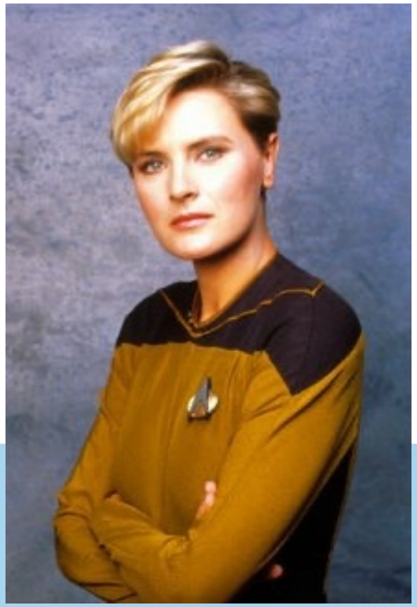
LTL Exercises

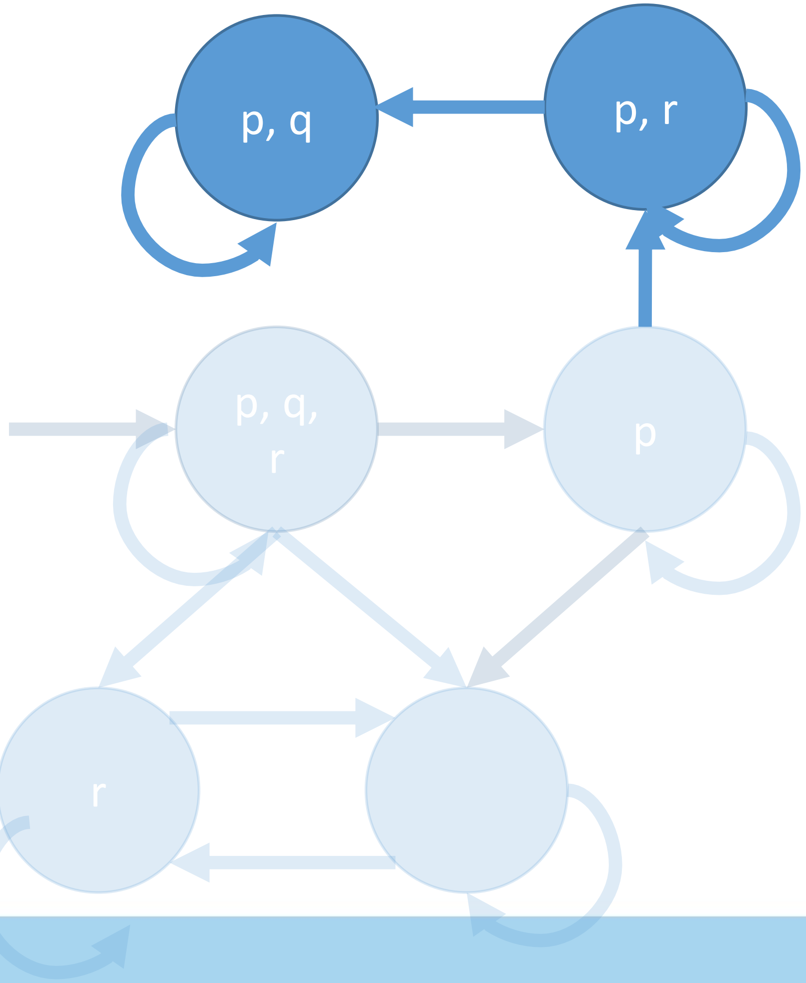
Example:

p = Yar alive

q = Yar on our ship

r = transporter ready





Scenario: Yar transports onto enemy ship

Yar transports back

Again, transporter takes time to recharge

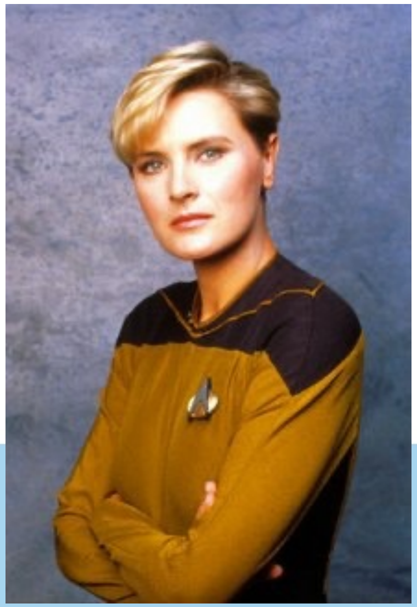
LTL Exercises

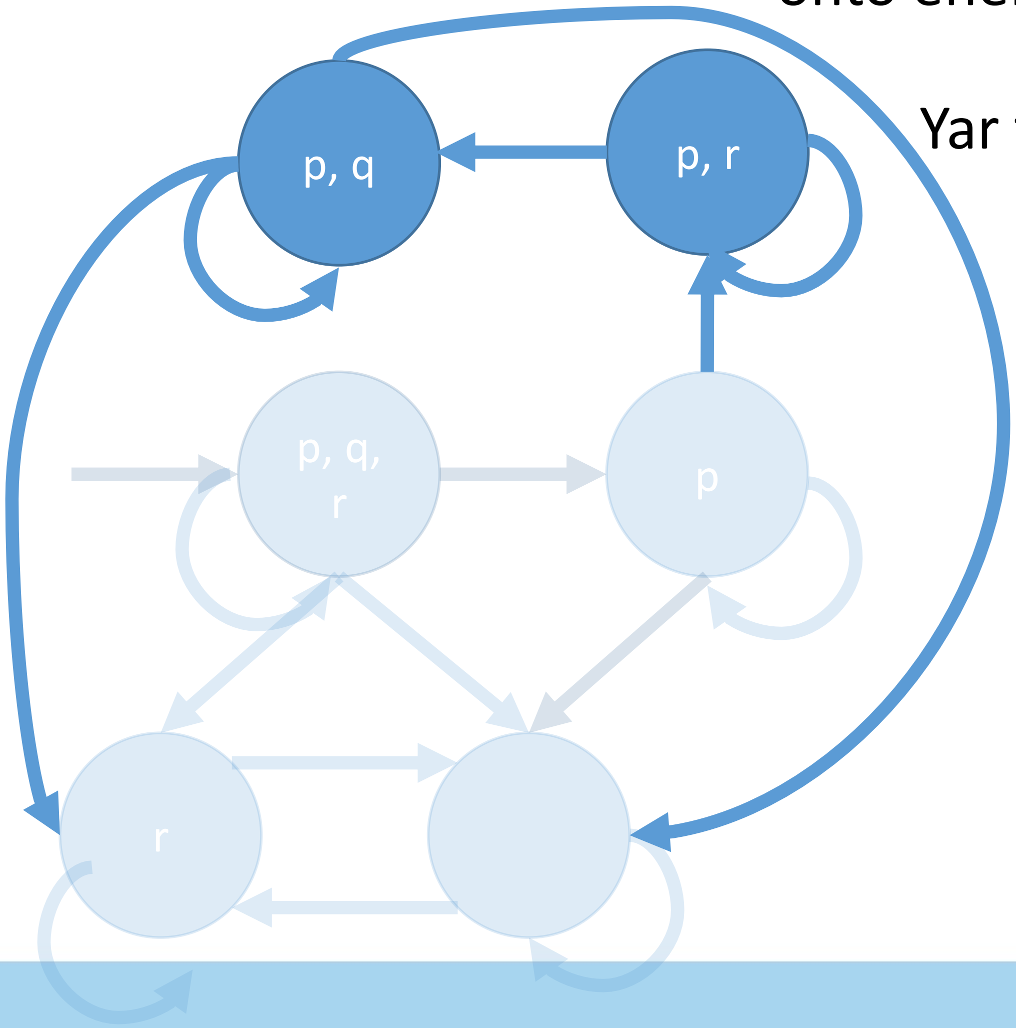
Example:

p = Yar alive

q = Yar on our ship

r = transporter ready





Scenario: Yar transports onto enemy ship

Yar transports back

Again, ship could be hit (before or after the transporter finishes charging)

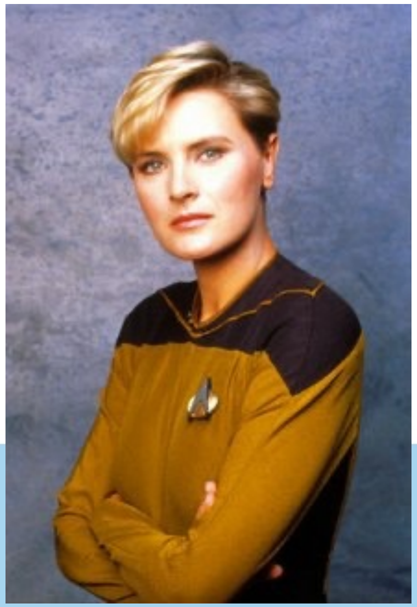
LTL Exercises

Example:

p = Yar alive

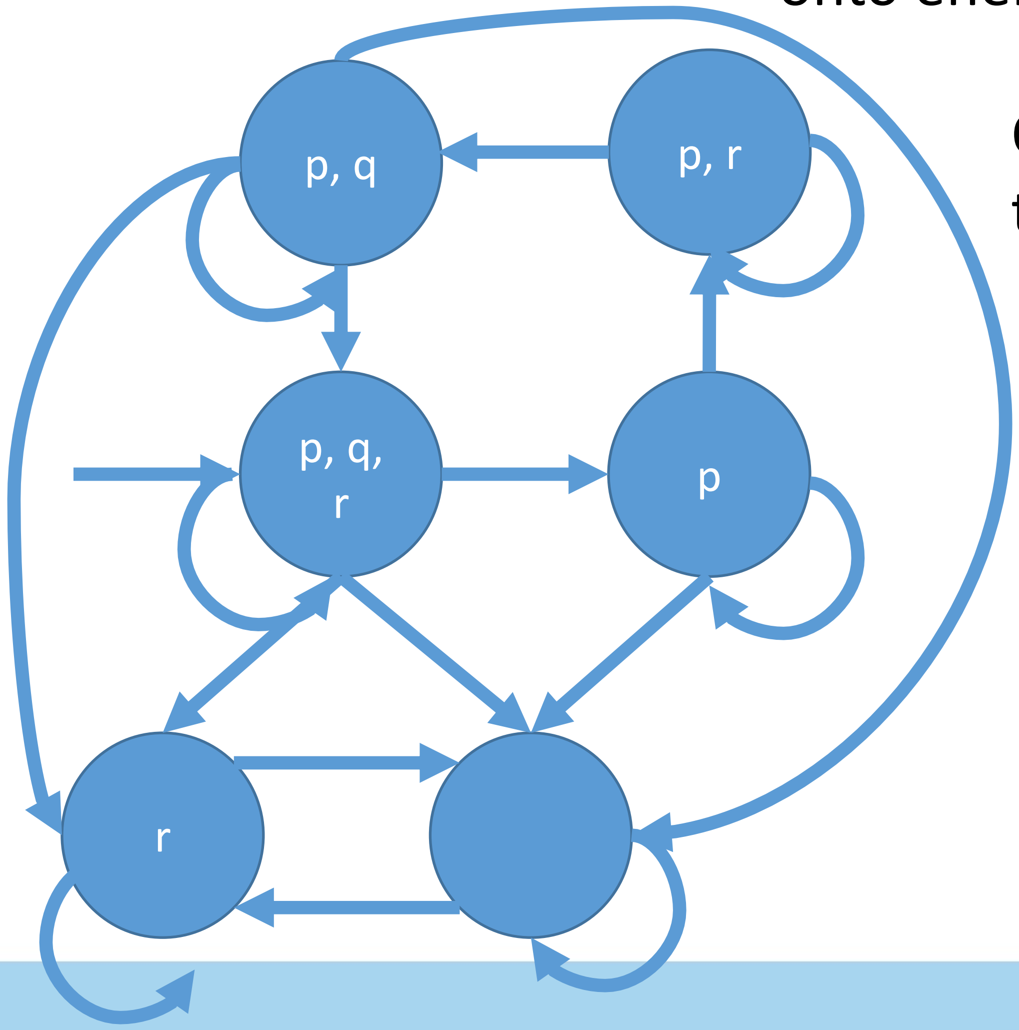
q = Yar on our ship

r = transporter ready



Scenario: Yar transports onto enemy ship

Otherwise, return to starting state



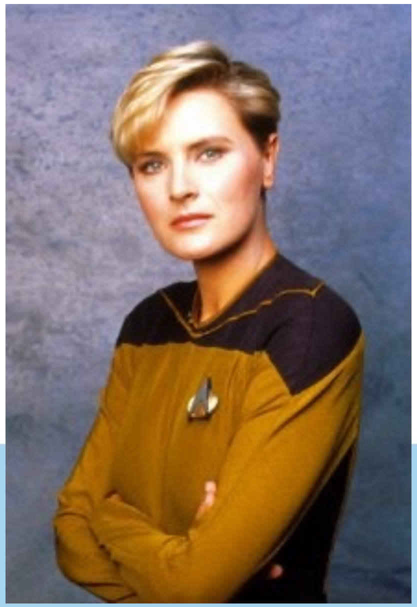
LTL Exercises

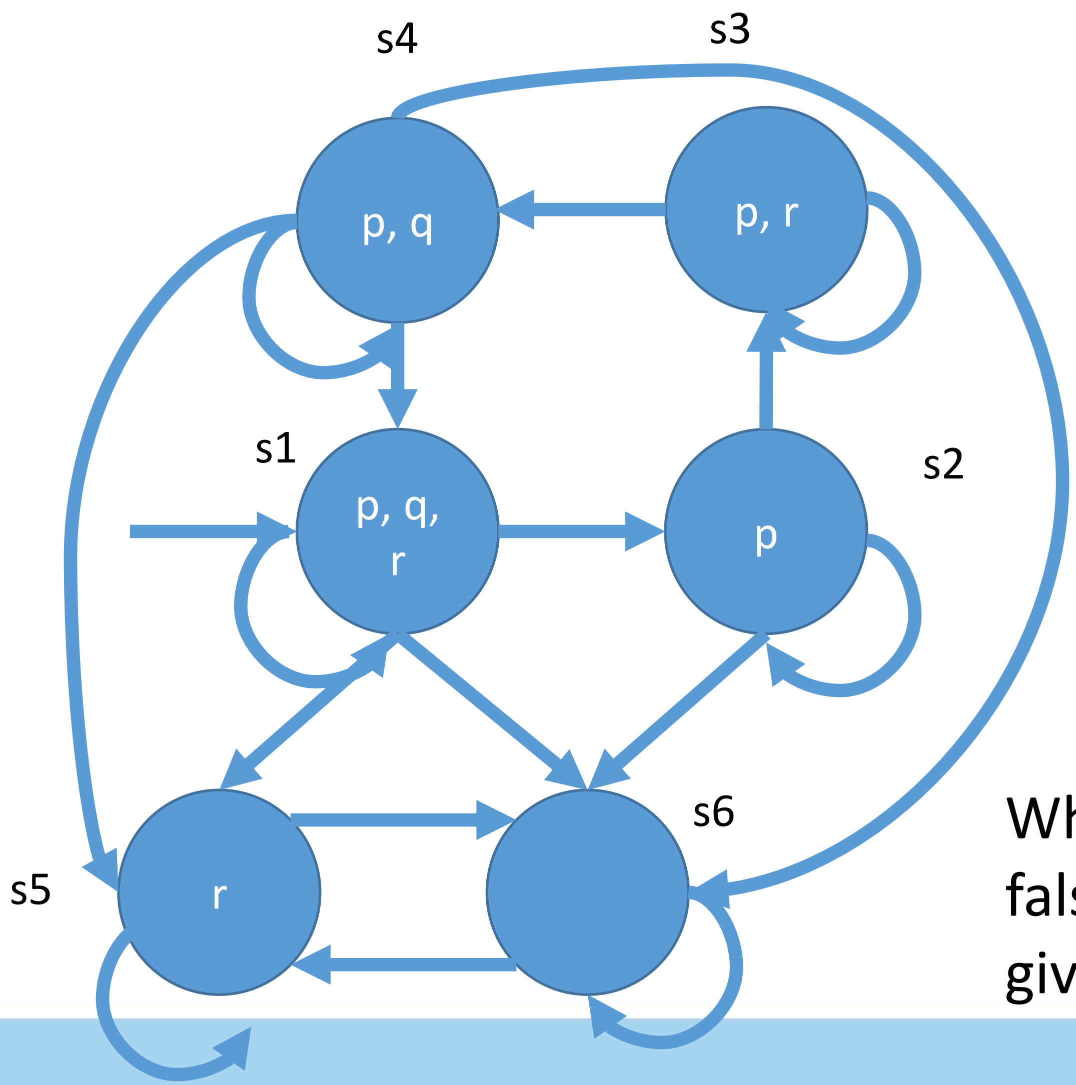
Example:

p = Yar alive

q = Yar on our ship

r = transporter ready





What are some true and false statements in LTL, given a path?

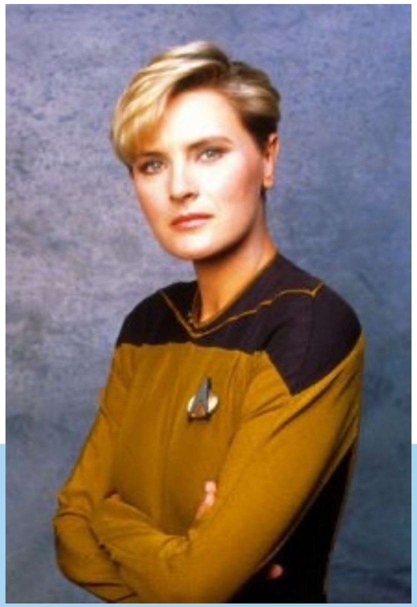
LTL Exercises

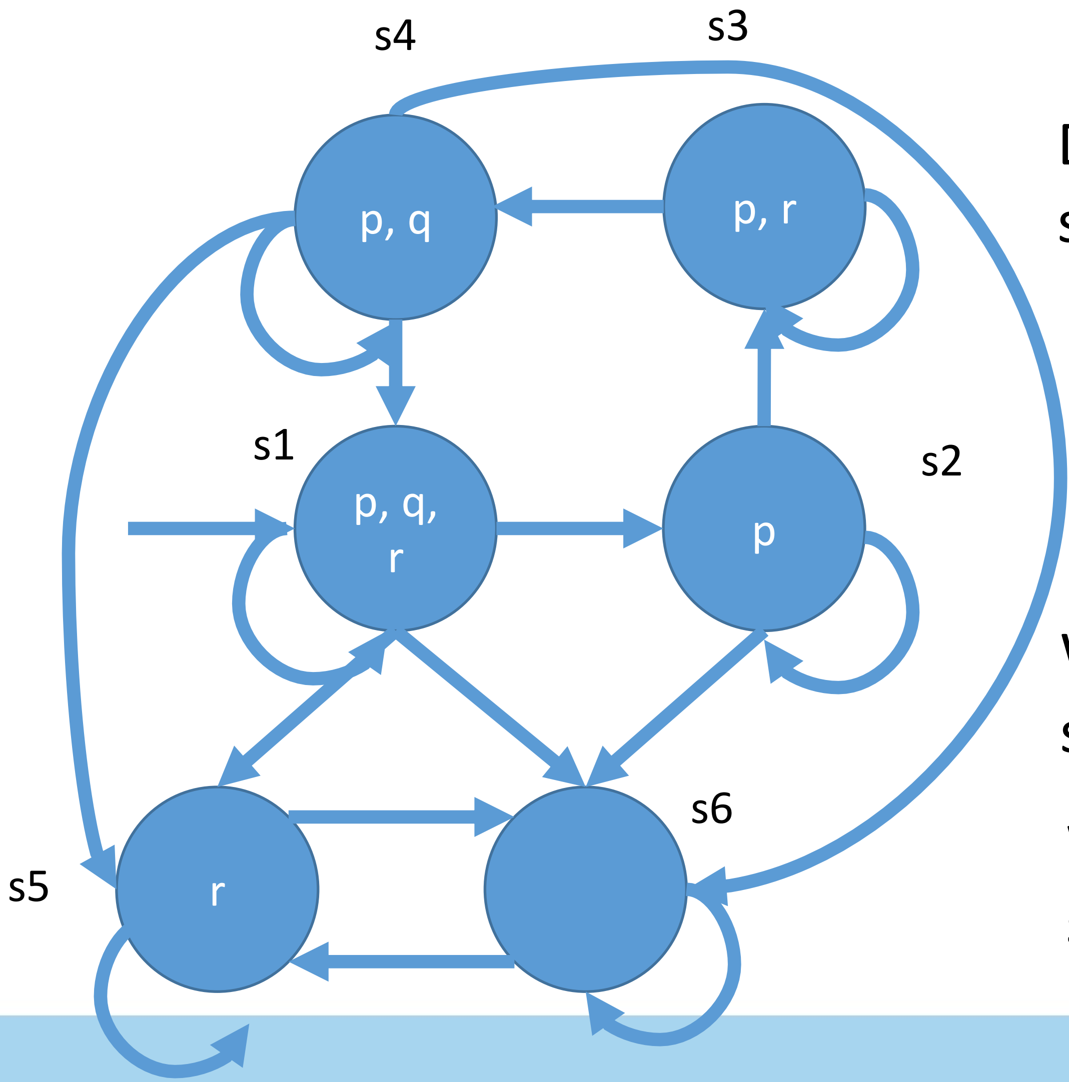
Example:

p = Yar alive

q = Yar on our ship

r = transporter ready





Things to note/discuss:

Do we ever get stuck?

8 possible states, but this diagram only shows 6. Why?

Why no edge from s2 to s1?

Why no edge from s3 to s5?

Observed data vs. mechanism

Why do we not assume that Yar can die on an enemy ship, even when the transporter is ready?

- Mechanistic?
    - Maybe our system automatically transports her back
- Observationally?
    - Maybe we never observe her dying when the transporter is ready

Idea; there *could* be an edge; we currently don't know the difference between the system and our modeling choice

Model design informed by LTL usage

- LTL for checking state sequences
  - Given a trace, do certain properties hold?
    - Does Yar survive the encounter?
    - Does Yar return to our ship?
- LTL for specifying goals
  - What properties do we want to hold?
  - Can we select actions to comport with goals?

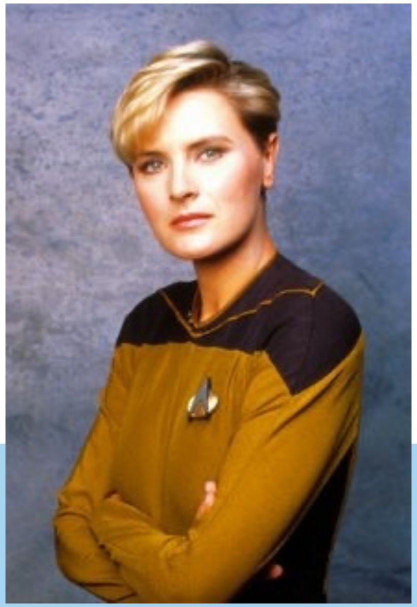
LTL Exercises

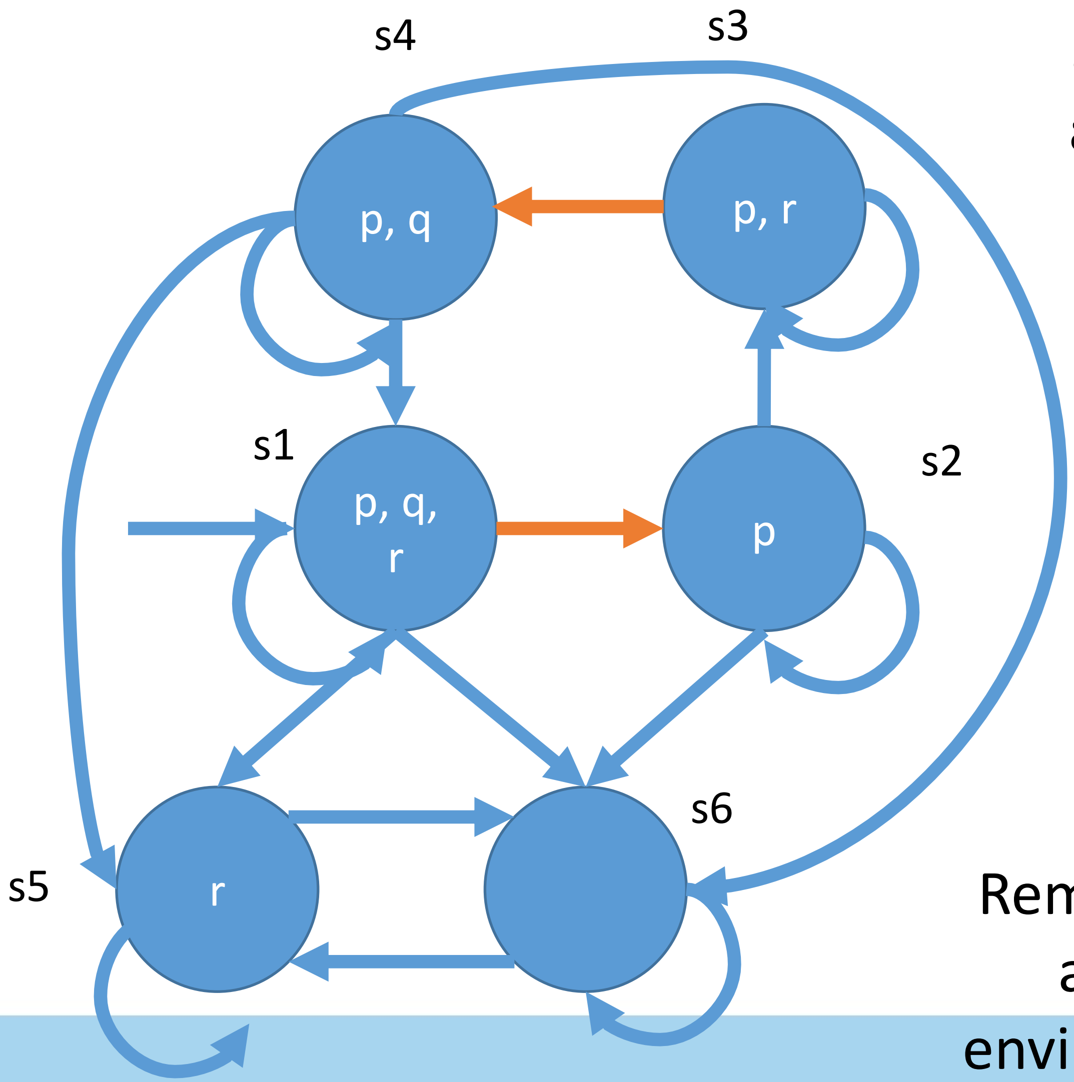
Example:

p = Yar alive

q = Yar on our ship

r = transporter ready





Some state transitions are associated with **actions** *we* take

Remaining transitions associated with environment dynamics

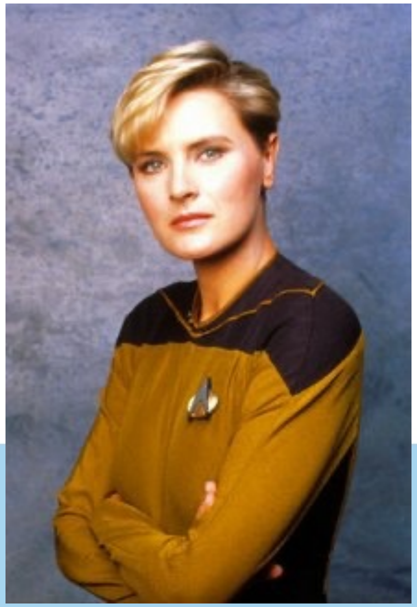
LTL Exercises

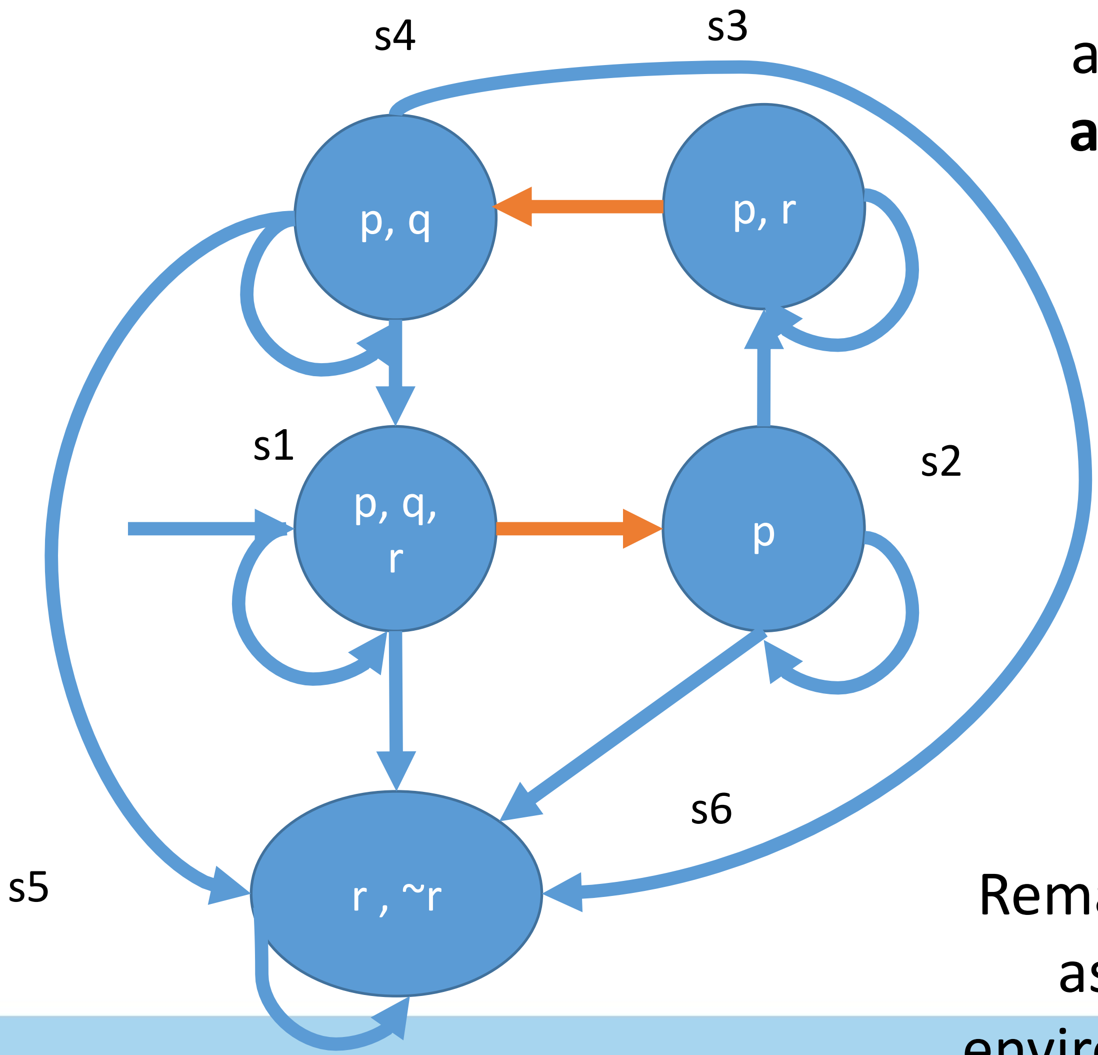
Example:

p = Yar alive

q = Yar on our ship

r = transporter ready





Some state transitions are associated with **actions** *we* take

Remaining transitions associated with environment dynamics

Model checking

Recall, a model satisfies a formula iff all paths satisfy the formula

Model checking approach:

1. Let a trace be a sequence of atoms and their values. A trace can be constructed from a path
2. Given a formula we want satisfied, construct an automaton such that if the automaton ***does not satisfy*** the formula, the automaton **accepts** the trace.
3. Walk through both the model and the automaton in tandem; if the automaton accepts a trace, then the model *does not satisfy* the formula.

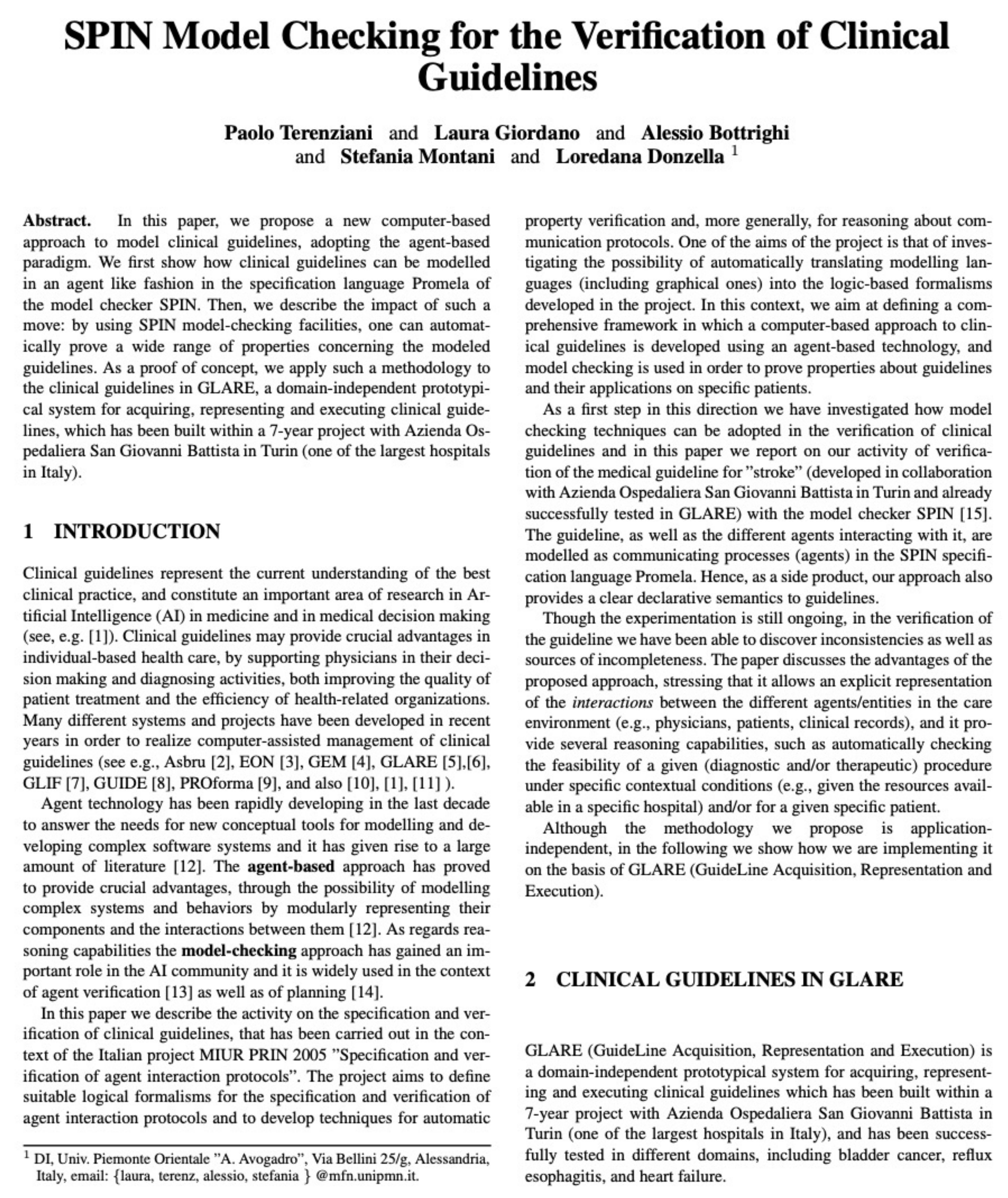
Model checking software

NuSMV

MR

Spin

checked



SPIN Model Checking for the Verification of Clinical Guidelines

Paolo Terenziani and **Laura Giordano** and **Alessio Bottrighi** and **Stefania Montani** and **Loredana Donzella** [1]

Abstract. In this paper, we propose a new computer-based approach to model clinical guidelines, adopting the agent-based paradigm. We first show how clinical guidelines can be modelled in an agent like fashion in the specification language Promela of the model checker SPIN. Then, we describe the impact of such a move: by using SPIN model-checking facilities, one can automatically prove a wide range of properties concerning the modeled guidelines. As a proof of concept, we apply such a methodology to the clinical guidelines in GLARE, a domain-independent prototypical system for acquiring, representing and executing clinical guidelines, which has been built within a 7-year project with Azienda Ospedaliera San Giovanni Battista in Turin (one of the largest hospitals in Italy).

1 INTRODUCTION

Clinical guidelines represent the current understanding of the best clinical practice, and constitute an important area of research in Artificial Intelligence (AI) in medicine and in medical decision making (see, e.g. [1]). Clinical guidelines may provide crucial advantages in individual-based health care, by supporting physicians in their decision making and diagnosing activities, both improving the quality of patient treatment and the efficiency of health-related organizations. Many different systems and projects have been developed in recent years in order to realize computer-assisted management of clinical guidelines (see e.g., Asbru [2], EON [3], GEM [4], GLARE [5],[6], GLIF [7], GUIDE [8], PROforma [9], and also [10], [1], [11]).

Agent technology has been rapidly developing in the last decade to answer the needs for new conceptual tools for modelling and developing complex software systems and it has given rise to a large amount of literature [12]. The **agent-based** approach has proved to provide crucial advantages, through the possibility of modelling complex systems and behaviors by modularly representing their components and the interactions between them [12]. As regards reasoning capabilities the **model-checking** approach has gained an important role in the AI community and it is widely used in the context of agent verification [13] as well as of planning [14].

In this paper we describe the activity on the specification and verification of clinical guidelines, that has been carried out in the context of the Italian project MIUR PRIN 2005 "Specification and verification of agent interaction protocols". The project aims to define suitable logical formalisms for the specification and verification of agent interaction protocols and to develop techniques for automatic property verification and, more generally, for reasoning about communication protocols. One of the aims of the project is that of investigating the possibility of automatically translating modelling languages (including graphical ones) into the logic-based formalisms developed in the project. In this context, we aim at defining a comprehensive framework in which a computer-based approach to clinical guidelines is developed using an agent-based technology, and model checking is used in order to prove properties about guidelines and their applications on specific patients.

As a first step in this direction we have investigated how model checking techniques can be adopted in the verification of clinical guidelines and in this paper we report on our activity of verification of the medical guideline for "stroke" (developed in collaboration with Azienda Ospedaliera San Giovanni Battista in Turin and already successfully tested in GLARE) with the model checker SPIN [15]. The guideline, as well as the different agents interacting with it, are modelled as communicating processes (agents) in the SPIN specification language Promela. Hence, as a side product, our approach also provides a clear declarative semantics to guidelines.

Though the experimentation is still ongoing, in the verification of the guideline we have been able to discover inconsistencies as well as sources of incompleteness. The paper discusses the advantages of the proposed approach, stressing that it allows an explicit representation of the *interactions* between the different agents/entities in the care environment (e.g., physicians, patients, clinical records), and it provide several reasoning capabilities, such as automatically checking the feasibility of a given (diagnostic and/or therapeutic) procedure under specific contextual conditions (e.g., given the resources available in a specific hospital) and/or for a given specific patient.

Although the methodology we propose is application-independent, in the following we show how we are implementing it on the basis of GLARE (GuideLine Acquisition, Representation and Execution).

2 CLINICAL GUIDELINES IN GLARE

GLARE (GuideLine Acquisition, Representation and Execution) is a domain-independent prototypical system for acquiring, representing and executing clinical guidelines which has been built within a 7-year project with Azienda Ospedaliera San Giovanni Battista in Turin (one of the largest hospitals in Italy), and has been successfully tested in different domains, including bladder cancer, reflux esophagitis, and heart failure.

[1] DI, Univ. Piemonte Orientale "A. Avogadro", Via Bellini 25/g, Alessandria, Italy, email: {laura, terenz, alessio, stefania } @mfn.unipmn.it.

Application: Medicine

- Verification of medical protocols, e.g., treatment guidelines
- Models interaction between physicism and ontology consisting of:
    - Patient information
    - Treatment protocols
    - Current state

Decoy Allocation Games on Graphs with Temporal Logic Objectives

Abhishek N. Kulkarni[1][0000–0002–1083–8507], Jie Fu[1][0000–0002–4470–2827], Huan Luo[1][0000–0002–1578–9409], Charles A. Kamhoua[2][0000–0003–2169–5975], and Nandi O. Leslie[2][0000–0001–5888–8784]

[1] Worceter Polytechnic Institute, Worcester MA 01609, USA
{ankulkarni,jfu2}@wpi.edu, hluo12@126.com
[2] U.S. Army Research Laboratory, Adelphi, MD 20783, USA
{charles.a.kamhoua.civ,nandi.o.leslie.ctr}@mail.mil

Abstract. We study a class of games, in which the adversary (attacker) is to satisfy a complex mission specified in linear temporal logic, and the defender is to prevent the adversary from achieving its goal. A deceptive defender can allocate decoys, in addition to defense actions, to create disinformation for the attacker. Thus, we focus on the problem of jointly synthesizing a decoy placement strategy and a deceptive defense strategy that maximally exploits the incomplete information the attacker about the decoy locations. We introduce a model of hypergames on graphs with temporal logic objectives to capture such adversarial interactions with asymmetric information. Using the hypergame model, we analyze the effectiveness of a given decoy placement, quantified by the set of deceptive winning states where the defender can prevent the attacker from satisfying the attack objective given its incomplete information about decoy locations. Then, we investigate how to place decoys to maximize the defender's deceptive winning region. Considering the large search space for all possible decoy allocation strategies, we incorporate the idea of compositional synthesis from formal methods and show that the objective function in the class of decoy allocation problem is monotone and non-decreasing. We derive the sufficient conditions under which the objective function for the decoy allocation problem is submodular, or supermodular, respectively. We show a sub-optimal allocation can be efficiently computed by iteratively composing the solutions of hypergames with a subset of decoys and the solution of a hypergame given a single decoy. We use a running example to illustrate the proposed method.

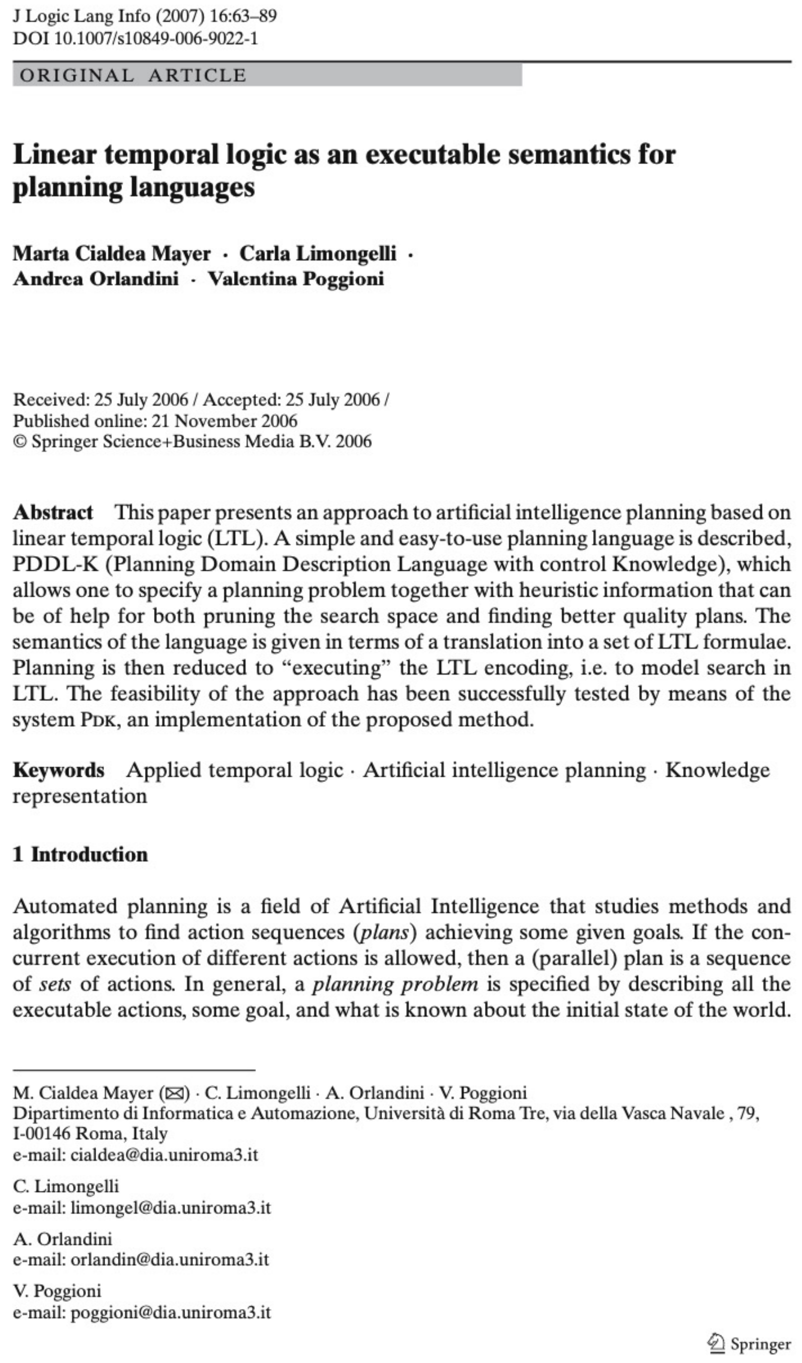
Keywords: Games on Graphs · Hypergames · Deception · Temporal Logic

1 Introduction

In security and defense applications, deception plays a key role to mitigate the information and strategic disadvantages of the defender against adversaries. In this paper, we investigate the design of active defense with deception for a class

Application: Security

- Verification of defense against a threat model over time
- Models interaction between a defense strategy (allocating decoys in a networked environment) and adversarial behavior



J Logic Lang Info (2007) 16:63–89
DOI 10.1007/s10849-006-9022-1

ORIGINAL ARTICLE

Linear temporal logic as an executable semantics for planning languages

Marta Cialdea Mayer · Carla Limongelli · Andrea Orlandini · Valentina Poggioni

Received: 25 July 2006 / Accepted: 25 July 2006 /
Published online: 21 November 2006
© Springer Science+Business Media B.V. 2006

Abstract This paper presents an approach to artificial intelligence planning based on linear temporal logic (LTL). A simple and easy-to-use planning language is described, PDDL-K (Planning Domain Description Language with control Knowledge), which allows one to specify a planning problem together with heuristic information that can be of help for both pruning the search space and finding better quality plans. The semantics of the language is given in terms of a translation into a set of LTL formulae. Planning is then reduced to "executing" the LTL encoding, i.e. to model search in LTL. The feasibility of the approach has been successfully tested by means of the system Pdk, an implementation of the proposed method.

Keywords Applied temporal logic · Artificial intelligence planning · Knowledge representation

1 Introduction

Automated planning is a field of Artificial Intelligence that studies methods and algorithms to find action sequences (*plans*) achieving some given goals. If the concurrent execution of different actions is allowed, then a (parallel) plan is a sequence of *sets* of actions. In general, a *planning problem* is specified by describing all the executable actions, some goal, and what is known about the initial state of the world.

M. Cialdea Mayer (✉) · C. Limongelli · A. Orlandini · V. Poggioni
Dipartimento di Informatica e Automazione, Università di Roma Tre, via della Vasca Navale , 79, I-00146 Roma, Italy
e-mail: cialdea@dia.uniroma3.it

C. Limongelli
e-mail: limongel@dia.uniroma3.it

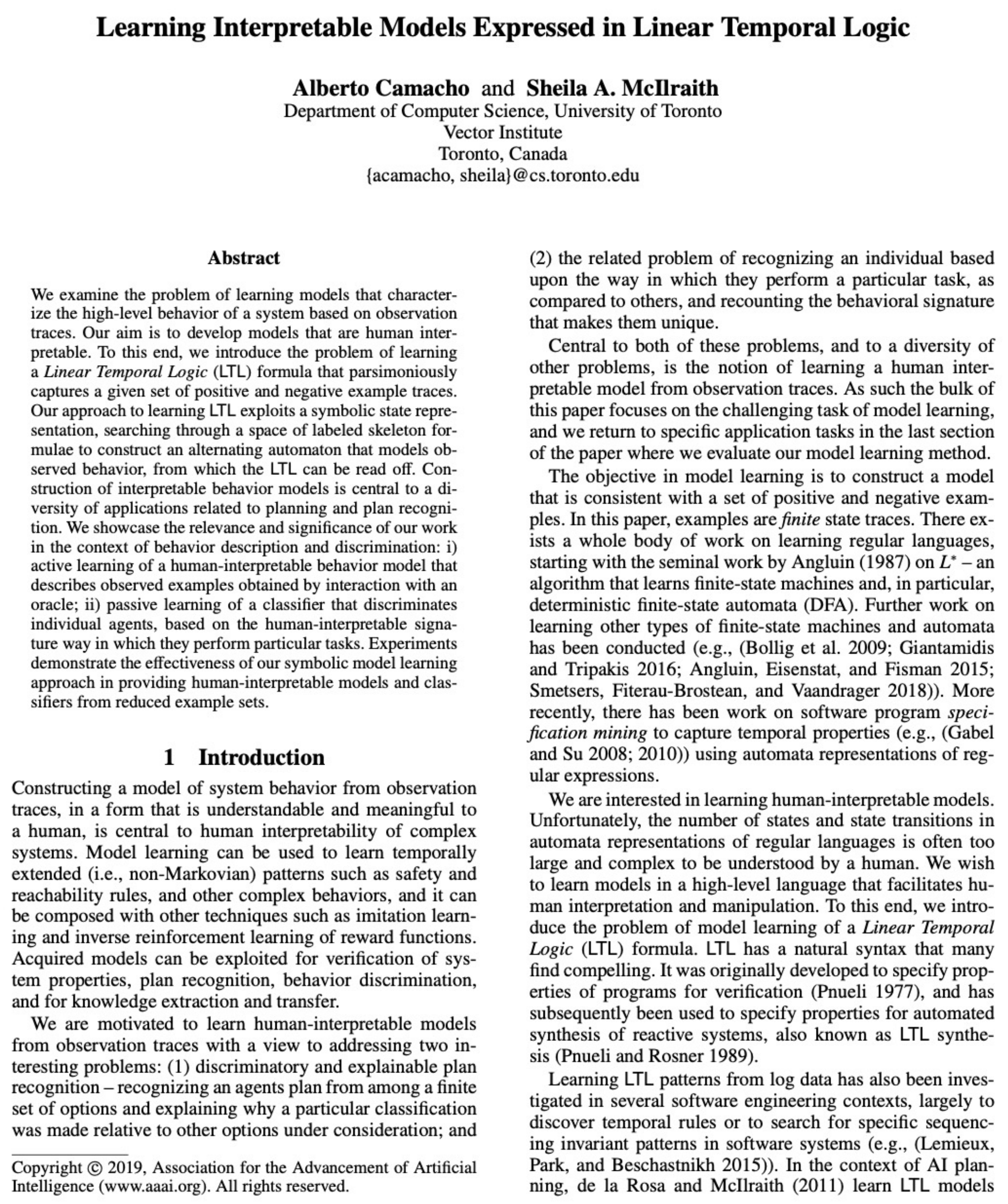
A. Orlandini
e-mail: orlandin@dia.uniroma3.it

V. Poggioni
e-mail: poggioni@dia.uniroma3.it

Springer

Application: AI Planning

- Classical planning does not capture time
- Use LTL to temporally extend actions
- Use a high-level language (e.g., PDDL) to encode plans
- Compile to LTL to be consistent with both action and environment dynamics



Learning Interpretable Models Expressed in Linear Temporal Logic

Alberto Camacho and **Sheila A. McIlraith**
Department of Computer Science, University of Toronto
Vector Institute
Toronto, Canada
{acamacho, sheila}@cs.toronto.edu

Abstract

We examine the problem of learning models that characterize the high-level behavior of a system based on observation traces. Our aim is to develop models that are human interpretable. To this end, we introduce the problem of learning a *Linear Temporal Logic* (LTL) formula that parsimoniously captures a given set of positive and negative example traces. Our approach to learning LTL exploits a symbolic state representation, searching through a space of labeled skeleton formulae to construct an alternating automaton that models observed behavior, from which the LTL can be read off. Construction of interpretable behavior models is central to a diversity of applications related to planning and plan recognition. We showcase the relevance and significance of our work in the context of behavior description and discrimination: i) active learning of a human-interpretable behavior model that describes observed examples obtained by interaction with an oracle; ii) passive learning of a classifier that discriminates individual agents, based on the human-interpretable signature way in which they perform particular tasks. Experiments demonstrate the effectiveness of our symbolic model learning approach in providing human-interpretable models and classifiers from reduced example sets.

1 Introduction

Constructing a model of system behavior from observation traces, in a form that is understandable and meaningful to a human, is central to human interpretability of complex systems. Model learning can be used to learn temporally extended (i.e., non-Markovian) patterns such as safety and reachability rules, and other complex behaviors, and it can be composed with other techniques such as imitation learning and inverse reinforcement learning of reward functions. Acquired models can be exploited for verification of system properties, plan recognition, behavior discrimination, and for knowledge extraction and transfer.

We are motivated to learn human-interpretable models from observation traces with a view to addressing two interesting problems: (1) discriminatory and explainable plan recognition – recognizing an agents plan from among a finite set of options and explaining why a particular classification was made relative to other options under consideration; and (2) the related problem of recognizing an individual based upon the way in which they perform a particular task, as compared to others, and recounting the behavioral signature that makes them unique.

Central to both of these problems, and to a diversity of other problems, is the notion of learning a human interpretable model from observation traces. As such the bulk of this paper focuses on the challenging task of model learning, and we return to specific application tasks in the last section of the paper where we evaluate our model learning method.

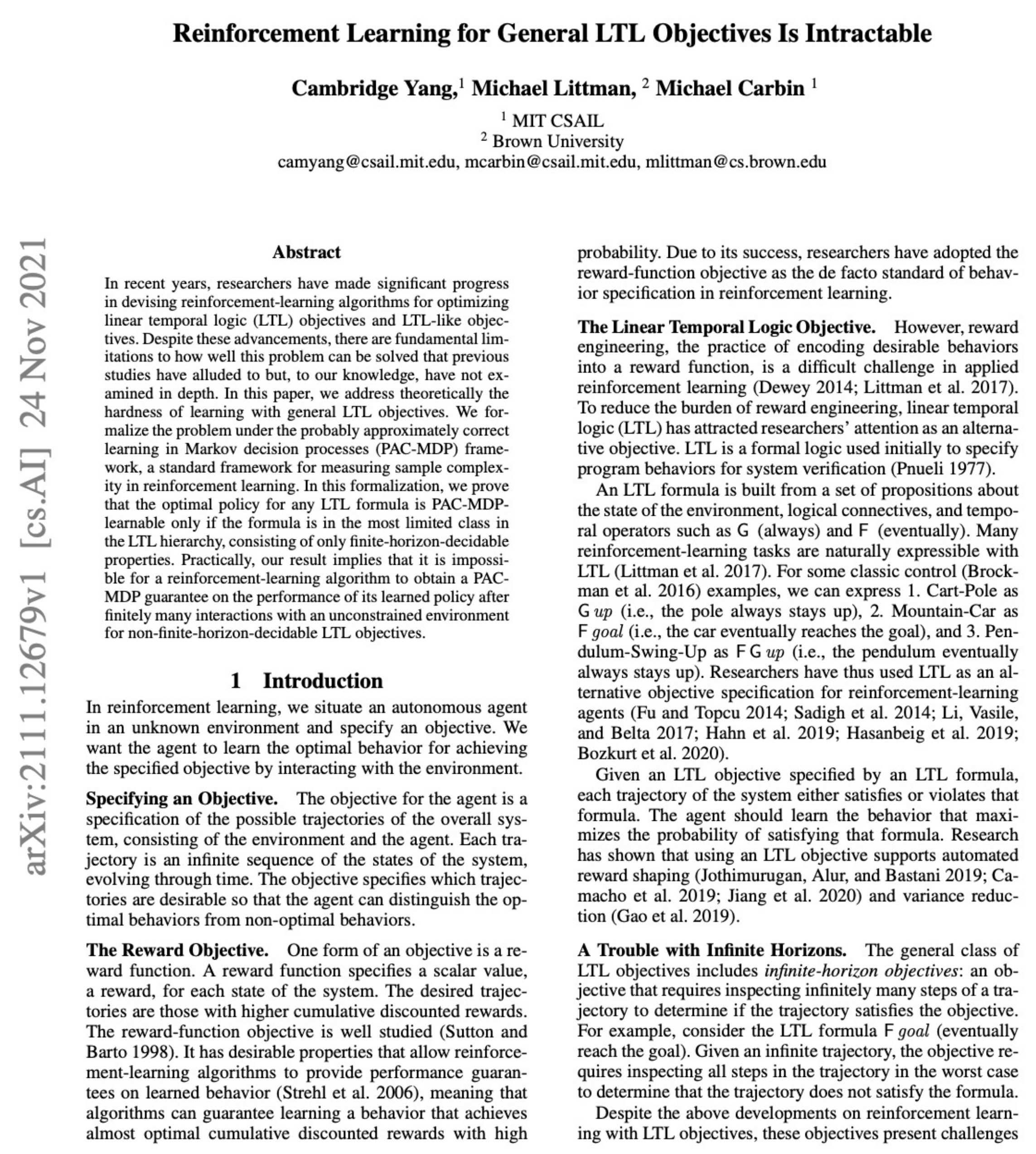
The objective in model learning is to construct a model that is consistent with a set of positive and negative examples. In this paper, examples are *finite* state traces. There exists a whole body of work on learning regular languages, starting with the seminal work by Angluin (1987) on L^* – an algorithm that learns finite-state machines and, in particular, deterministic finite-state automata (DFA). Further work on learning other types of finite-state machines and automata has been conducted (e.g., (Bollig et al. 2009; Giantamidis and Tripakis 2016; Angluin, Eisenstat, and Fisman 2015; Smetsers, Fiterau-Brostean, and Vaandrager 2018)). More recently, there has been work on software program *specification mining* to capture temporal properties (e.g., (Gabel and Su 2008; 2010)) using automata representations of regular expressions.

We are interested in learning human-interpretable models. Unfortunately, the number of states and state transitions in automata representations of regular languages is often too large and complex to be understood by a human. We wish to learn models in a high-level language that facilitates human interpretation and manipulation. To this end, we introduce the problem of model learning of a *Linear Temporal Logic* (LTL) formula. LTL has a natural syntax that many find compelling. It was originally developed to specify properties of programs for verification (Pnueli 1977), and has subsequently been used to specify properties for automated synthesis of reactive systems, also known as LTL synthesis (Pnueli and Rosner 1989).

Learning LTL patterns from log data has also been investigated in several software engineering contexts, largely to discover temporal rules or to search for specific sequencing invariant patterns in software systems (e.g., (Lemieux, Park, and Beschastnikh 2015)). In the context of AI planning, de la Rosa and McIlraith (2011) learn LTL models

Application: AI interpretability

- Idea: observe sequences of state
- Learn LTL formulas to capture system invariants over time
- Claim: Can be used to generate interpretable policies over complex software systems



Reinforcement Learning for General LTL Objectives Is Intractable

Cambridge Yang,[1] Michael Littman, [2] Michael Carbin [1]
[1] MIT CSAIL
[2] Brown University
camyang@csail.mit.edu, mcarbin@csail.mit.edu, mlittman@cs.brown.edu

Abstract

In recent years, researchers have made significant progress in devising reinforcement-learning algorithms for optimizing linear temporal logic (LTL) objectives and LTL-like objectives. Despite these advancements, there are fundamental limitations to how well this problem can be solved that previous studies have alluded to but, to our knowledge, have not examined in depth. In this paper, we address theoretically the hardness of learning with general LTL objectives. We formalize the problem under the probably approximately correct learning in Markov decision processes (PAC-MDP) framework, a standard framework for measuring sample complexity in reinforcement learning. In this formalization, we prove that the optimal policy for any LTL formula is PAC-MDP-learnable only if the formula is in the most limited class in the LTL hierarchy, consisting of only finite-horizon-decidable properties. Practically, our result implies that it is impossible for a reinforcement-learning algorithm to obtain a PAC-MDP guarantee on the performance of its learned policy after finitely many interactions with an unconstrained environment for non-finite-horizon-decidable LTL objectives.

1 Introduction

In reinforcement learning, we situate an autonomous agent in an unknown environment and specify an objective. We want the agent to learn the optimal behavior for achieving the specified objective by interacting with the environment.

Specifying an Objective. The objective for the agent is a specification of the possible trajectories of the overall system, consisting of the environment and the agent. Each trajectory is an infinite sequence of the states of the system, evolving through time. The objective specifies which trajectories are desirable so that the agent can distinguish the optimal behaviors from non-optimal behaviors.

The Reward Objective. One form of an objective is a reward function. A reward function specifies a scalar value, a reward, for each state of the system. The desired trajectories are those with higher cumulative discounted rewards. The reward-function objective is well studied (Sutton and Barto 1998). It has desirable properties that allow reinforcement-learning algorithms to provide performance guarantees on learned behavior (Strehl et al. 2006), meaning that algorithms can guarantee learning a behavior that achieves almost optimal cumulative discounted rewards with high probability. Due to its success, researchers have adopted the reward-function objective as the de facto standard of behavior specification in reinforcement learning.

The Linear Temporal Logic Objective. However, reward engineering, the practice of encoding desirable behaviors into a reward function, is a difficult challenge in applied reinforcement learning (Dewey 2014; Littman et al. 2017). To reduce the burden of reward engineering, linear temporal logic (LTL) has attracted researchers' attention as an alternative objective. LTL is a formal logic used initially to specify program behaviors for system verification (Pnueli 1977).

An LTL formula is built from a set of propositions about the state of the environment, logical connectives, and temporal operators such as G (always) and F (eventually). Many reinforcement-learning tasks are naturally expressible with LTL (Littman et al. 2017). For some classic control (Brockman et al. 2016) examples, we can express 1. Cart-Pole as G *up* (i.e., the pole always stays up), 2. Mountain-Car as F *goal* (i.e., the car eventually reaches the goal), and 3. Pendulum-Swing-Up as F G *up* (i.e., the pendulum eventually always stays up). Researchers have thus used LTL as an alternative objective specification for reinforcement-learning agents (Fu and Topcu 2014; Sadigh et al. 2014; Li, Vasile, and Belta 2017; Hahn et al. 2019; Hasanbeig et al. 2019; Bozkurt et al. 2020).

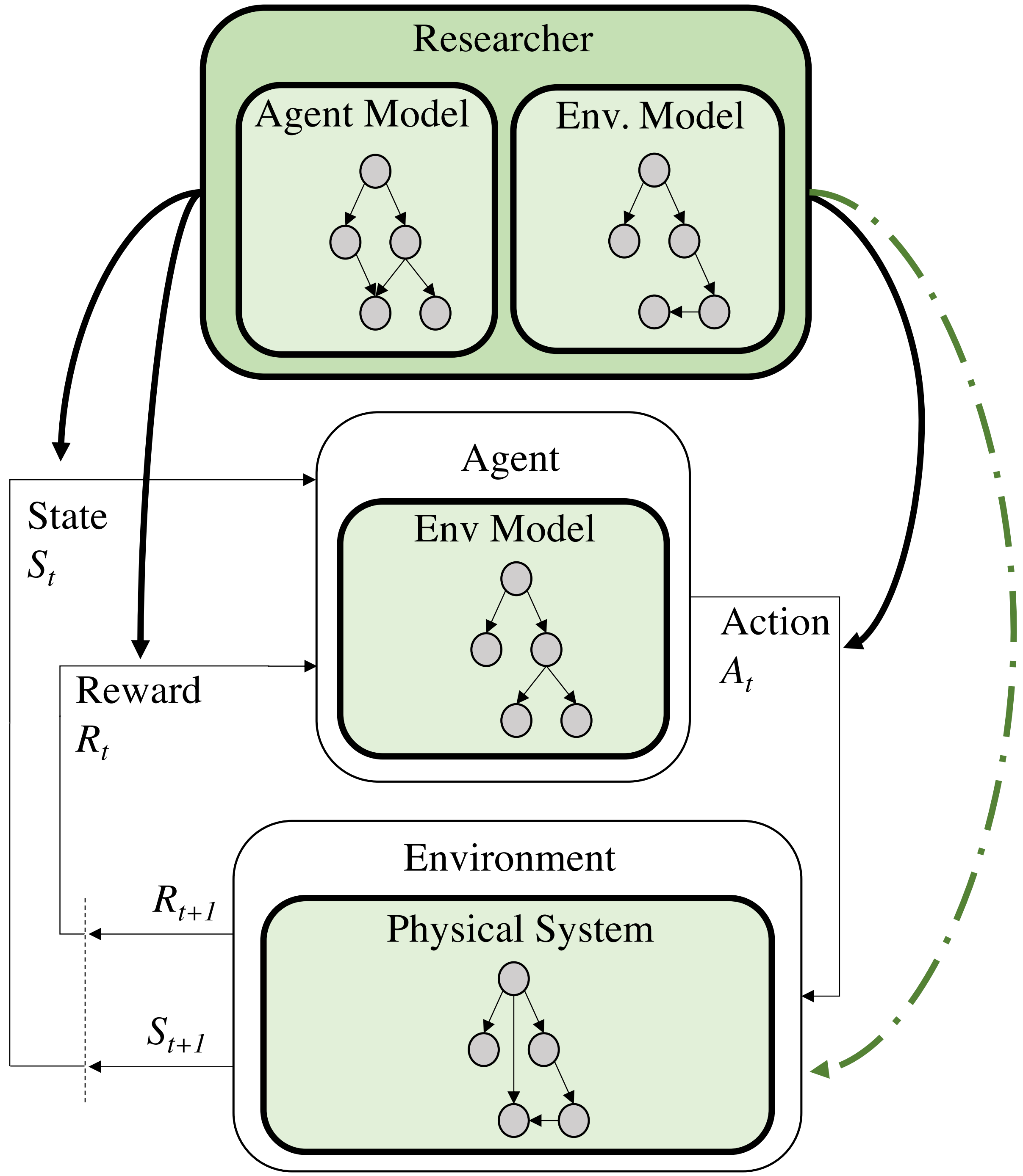
Given an LTL objective specified by an LTL formula, each trajectory of the system either satisfies or violates that formula. The agent should learn the behavior that maximizes the probability of satisfying that formula. Research has shown that using an LTL objective supports automated reward shaping (Jothimurugan, Alur, and Bastani 2019; Camacho et al. 2019; Jiang et al. 2020) and variance reduction (Gao et al. 2019).

A Trouble with Infinite Horizons. The general class of LTL objectives includes *infinite-horizon objectives*: an objective that requires inspecting infinitely many steps of a trajectory to determine if the trajectory satisfies the objective. For example, consider the LTL formula F *goal* (eventually reach the goal). Given an infinite trajectory, the objective requires inspecting all steps in the trajectory in the worst case to determine that the trajectory does not satisfy the formula.

Despite the above developments on reinforcement learning with LTL objectives, these objectives present challenges

Intractability of LTL

- Utility-driven objectives can be expressed in LTL
- Problem: LTL formulae over paths
- Exponential blow-up not just in state space (known issue), but also over paths



What other problems might we face?