

Lattice Annotated Temporal (LAT) Logic for Non-Markovian Reasoning

KAUSTUV MUKHERJI, JAIKRISHNA MANOJKUMAR PATIL, DYUMAN ADITYA, and

PAULO SHAKARIAN, Syracuse University, USA

DEVENDRA PARKAR and LAHARI POKALA, Arizona State University, USA

CLARK DORMAN, Scientific Systems Company, Inc., USA

GERARDO I. SIMARI, Department of Computer Science and Engineering, Universidad Nacional del Sur (UNS) &

Institute for Computer Science and Engineering (ICIC UNS-CONICET), Argentina

We introduce Lattice Annotated Temporal (LAT) Logic, an extension of Generalized Annotated Logic Programs (GAPs) that incorporates temporal reasoning and supports open-world semantics through the use of a lower lattice structure. This logic combines an efficient deduction process with temporal logic programming to support non-Markovian relationships and open-world reasoning capabilities. The open-world aspect, a by-product of the use of the lower-lattice annotation structure, allows for efficient grounding through a Skolemization process, even in domains with infinite or highly diverse constants. We provide a suite of theoretical results that bound the computational complexity of the grounding process, in addition to showing that many of the results on GAPs (using an upper lattice) still hold with the lower lattice and temporal extensions (though different proof techniques are required). Our open-source implementation, PyReason, features modular design, machine-level optimizations, and direct integration with reinforcement learning environments. Empirical evaluations across multi-agent simulations and knowledge graph tasks demonstrate up to three orders of magnitude speedup and up to five orders of magnitude memory reduction while maintaining or improving task performance. Additionally, we evaluate LAT logic's value in reinforcement learning environments as a non-Markovian simulator, achieving up to three orders of magnitude faster simulation with improved agent performance, including a 26% increase in win rate due to capturing richer temporal dependencies. These results highlight LAT logic's potential as a unified, extensible framework for open world temporal reasoning in dynamic and uncertain environments. Our implementation is available at: pyreason.syracuse.edu.

CCS Concepts: • Theory of computation → Constraint and logic programming; Automated reasoning; Modal and temporal logics; Reinforcement learning.

Additional Key Words and Phrases: Logic programming, Generalized annotated program, Temporal logic, First-order logic, Open world reasoning, Reinforcement learning, Non-markovian dynamics.

ACM Reference Format:

Kaustuv Mukherji, Jaikrishna Manojkumar Patil, Dyuman Aditya,, Paulo Shakarian, Devendra Parkar, Lahari Pokala, Clark Dorman, and Gerardo I. Simari. 2018. Lattice Annotated Temporal (LAT) Logic for Non-Markovian Reasoning. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 43 pages. <https://doi.org/XXXXXX.XXXXXXXX>

Authors' Contact Information: [Kaustuv Mukherji](mailto:kmukherj@syr.edu), kmukherj@syr.edu; [Jaikrishna Manojkumar Patil](mailto:jpatil01@syr.edu), jpatil01@syr.edu; [Dyuman Aditya](mailto:daditya@syr.edu), daditya@syr.edu; [Paulo Shakarian](mailto:pashakar@syr.edu), pashakar@syr.edu, Syracuse University, Syracuse, New York, USA; [Devendra Parkar](mailto:dparkar1@asu.edu), dparkar1@asu.edu; [Lahari Pokala](mailto:lpokala@asu.edu), lpokala@asu.edu, Arizona State University, Tempe, Arizona, USA; Clark Dorman, clark.dorman@ssci.com, Scientific Systems Company, Inc., Woburn, Massachusetts, USA; [Gerardo I. Simari](mailto:gis@cs.uns.edu.ar), gis@cs.uns.edu.ar, Department of Computer Science and Engineering, Universidad Nacional del Sur (UNS) & Institute for Computer Science and Engineering (ICIC UNS-CONICET), Bahia Blanca, Argentina.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

Manuscript submitted to ACM

1 Introduction

Recent work in Artificial Intelligence (AI) has looked at the use of logic programming to bring explainability and robustness to modern AI systems [10, 21, 22]. Temporal logic programming [24, 26, 45, 60] provides the ability to directly model non-Markovian temporal relationships (i.e., the agent’s behavior can be dependent on multiple previous time steps) due to its ability to reason about if-then statements that span multiple time steps (see Figure 1 for an example logic program). This capability would make temporal logic programming a suitable candidate to replace standard Markov Decision Processes (MDPs) prevalently found in various reinforcement learning systems. However, temporal logic programming, such as APT logic [60], is intractable not only in terms of time but also space due to grounding, thereby limiting its applicability to such use-cases. In this paper, we overcome the problem with Lattice Annotated Temporal Logic or LAT logic. This logic is created by adding temporal extensions to a fragment of Generalized Annotated Programs (GAPs) [38], thereby permitting direct modeling of non-Markovian temporal relationships, which allows exact but tractable inference in a variety of cases. We retain the capabilities to represent uncertainty by virtue of the annotations, but leverage a lower-lattice structure (as opposed to an upper-lattice structure in GAPs), which not only enables open-world reasoning but also affords an efficient Skolemization process, enabling scalable grounding. We provide a theoretical foundation for LAT logic and a suite of experimental results, including a demonstration of effectiveness as a replacement for MDPs in reinforcement learning (RL) applications.

To make some of our ideas more concrete, we introduce our first running example, modeled after the use cases explored in our experiments. In Figure 2, we consider a simple example where two agents, one on foot patrol (shown with an icon of a person) and another in a patrol car (car icon) move in a geospatial area (marked by the square). The speed of the car is double that of the agent on foot. Figure 1 shows an excerpt from the logic program that governs the agents’ movements; note that in the syntax of annotated logic, truth values follow the atoms after a colon. While we will describe lower-lattice based truth values in the technical preliminaries, we note that $[1, 1]$ denotes truth, $[0, 0]$ denotes falsehood, and $[0, 1]$ denotes total uncertainty—any subset of the unit interval can be a truth value, allowing for expressions of various levels of first and second-order uncertainty. In Figure 1, the first two rules show how agents may move in different directions with different speeds, creating new constants in space when rules are fired. The last two rules help update the truth values to false when an agent moves away from the old location. Here, variable A is used for agents, while L_1, L_2 can be grounded with constants in space. The Δt is used here to capture the differing speeds between agents, showing how the temporal component can be used to capture the dynamics of a non-Markovian environment. Note that the Δt between antecedent and consequent can be heterogeneous within the logic program. For this example, at $t = 0$, the patrol car chooses to move left, and the foot patrol decides to go right. Following these action choices, the aforementioned rules are grounded and in effect two new constants are grounded in space, as shown in Figure 2 with red pins, after one and two timesteps.

To build the capabilities presented throughout the paper in these examples, we provide the following contributions:

- (1) We introduce LAT logic, an extension of Generalized Annotated Logic Programs (GAPs) that includes temporal extensions and leverages a lower lattice instead of an upper lattice -- a departure from the traditional GAPs on which it is based. We formalize the syntax and semantics and show how using a lower lattice for annotated logic facilitates open-world reasoning. We reprove results for satisfaction, consistency, and entailment for the extended logic with lower lattice and temporal aspects. Then we introduce the fixpoint operator that allows for deductive reasoning while mapping timepoint literal pairs to timepoint literal pairs, which readily supports non-Markovian reasoning. We show that the use of the lower lattice not only allows for open-world reasoning but

$$\Pi_{geo} = \{ at(A, L_2) : [1, 1] \xleftarrow[\Delta t=1]{} at(A, L_1) : [1, 1] \wedge moveLeft(A) : [1, 1] \wedge speed(A, fast) : [1, 1] \wedge left(L_1, L_2) : [1, 1], \\ \text{If fast-moving agent } A \text{ is at } L_1 \text{ and moves left, it will be at } L_2 \text{ (left of } L_1) \text{ after one time unit.} \\ at(A, L_2) : [1, 1] \xleftarrow[\Delta t=2]{} at(A, L_1) : [1, 1] \wedge moveRight(A) : [1, 1] \wedge speed(A, slow) : [1, 1] \wedge right(L_1, L_2) : [1, 1], \\ \text{If slow-moving agent } A \text{ is at } L_1 \text{ and moves right, it will be at } L_2 \text{ (right of } L_1) \text{ after two time units.} \\ at(A, L_1) : [0, 0] \xleftarrow[\Delta t=1]{} at(A, L_1) : [1, 1] \wedge moveLeft(A) : [1, 1], \\ \text{If agent } A \text{ is at } L_1 \text{ and moves left, it will no longer be at } L_1 \text{ after one time unit.} \\ at(A, L_1) : [0, 0] \xleftarrow[\Delta t=1]{} at(A, L_1) : [1, 1] \wedge moveRight(A) : [1, 1] \} \\ \text{If agent } A \text{ is at } L_1 \text{ and moves right, it will no longer be at } L_1 \text{ after one time unit.}$$

Fig. 1. Excerpt of logic program Π_{geo} for the geospatial example shown in Figure 2. English translations for each rule are also provided.

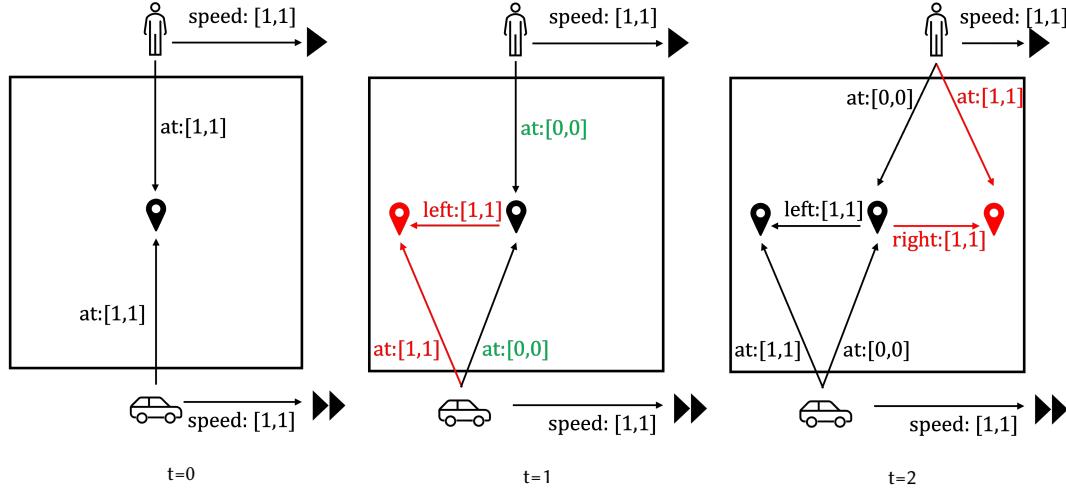


Fig. 2. Geospatial example: Creation of new atoms during inference for two time steps (left to right). Newly created atoms at each time point are shown in red. Existing atoms whose annotations change are marked in green.

also a natural form of Skolemization of constants and atoms that has applications to geospatial and knowledge graph reasoning tasks. We show that theoretical results on correctness for GAPs proven in [61] hold for our temporal extensions. In section 4.2, we analytically bound the creation of new ground atoms during reasoning, which suggests a significant reduction in grounding resulting from this form of Skolemization, and can provide significant speed-up in certain applications. Later, in Sections 6.1 and 6.2, we experimentally show that this result is a loose upper bound for practical domains where constants and atoms are diverse and sparse.

- (2) We describe an implementation of the logic fixpoint operator with highly optimized data structures that can leverage the Skolemization properties. The implementation is built on modern Python and is machine code optimized to handle graph-based data structures compatible with all common graph-based databases. Grounding

and inference processes are parallelized across threads to leverage computing resources optimally, thereby reducing running time. It is also modular, where new constants and rules can be added to the existing program at any point. Our temporal extensions also allow rules whose effects are observed at different intervals from when they are fired, and this directly supports building logic programs for non-Markovian dynamics. Among other applications, all of these capabilities make this implementation especially suitable for addressing common challenges seen in RL use cases. Our implementation, called PyReason, is available as an open-source project at <https://pyreason.syracuse.edu>. Various software and hardware acceleration techniques used are detailed in Section 5.5.

- (3) Our experimental evaluation encompasses two distinct applications: a multi-agent geospatial simulation and knowledge graph completion. The empirical results corroborate our theoretical analysis, demonstrating reductions of up to three orders of magnitude in the size of groundings for multi-step reasoning. Notably, while our theoretical analysis provides a conservative upper bound, practical observations show significantly fewer ground atoms with higher numbers of fixpoint applications. This results in enhanced scalability in both computational speed and memory efficiency, further improving the approach’s applicability in real-world scenarios.
- (4) We provide an extensive suite of experiments showing how Skolemization arising from the lower lattice of annotations provides multiple orders of magnitude of speedup when leveraged by efficient data structures present in our implementation. In Section 6.1, we show the efficacy of our Skolemization-based approach where we observe several orders of speedup - growing significantly with the number of ground atoms. Similarly, memory reduction is observed to increase with a higher number of ground atoms, giving up to five orders of magnitude of savings for the highest setting in our experiment. Section 6.2 shows the scaling capability of our approach on four popular knowledge bases: WN18RR and FB15k-237 [8], YAGO03-10 [65], and UMLS [46], with varying numbers of rules. Our approach is shown to always provide a speedup and memory reduction, which slowly settles towards the base resources as large number of constants, rules, and reasoning steps slowly moves towards a fully connected network. We also do some initial exploration of the potential of multi-step reasoning by showing how two-step reasoning can provide better results on information retrieval tasks across subsets of all four datasets.
- (5) Finally, in Section 6.3 we present a suite of experimental results where we leverage both the speedups described above along with the temporal characteristics of the logic to allow for efficient non-Markovian simulation of an environment for use in training a reinforcement learning agent. We show up to three orders of magnitude speedup compared with two simulation environments while maintaining agent performance. When non-Markovian capabilities are used, we show a notable 26% improvement in agent win rate, compared to when limited by the Markov assumption.

The rest of the paper is organized as follows. In Section 2, we review some related, well-established logics, discuss the importance of capturing non-Markovian dynamics for modern applications, and other relevant work. In Section 3, we describe the syntax and semantics of LAT logic, as well as provide a running example, which is an excerpt from domains used in our experiments. In Section 4, we provide theoretical results on our temporal extensions to GAPs and resulting performance gains. We detail our open-source implementation of LAT logic in Section 5. Sections 6.1 and 6.2 contain two sets of experiments that verify the theoretical performance gains derived in Section 4.2, and show the scaling capability of our implementation on two diverse applications. In Section 6.3, we show how our implementation is amenable to be used effectively as a simulator for reinforcement learning applications. Finally, in Section 7, we summarize our findings and share our thoughts on future work.

2 Related Work

Logic programming emerged decades ago as a foundational paradigm in artificial intelligence, enabling the formal representation of knowledge through declarative statements and supporting automated reasoning via deductive inference [39]. This approach facilitated the systematic derivation of consequences from encoded knowledge, significantly advancing the capabilities of early AI systems. Common implementations of logic programming like Prolog [19] and Datalog [13] are designed for non-temporal applications and do not support uncertainty or annotations – While it is always possible to devise special syntax (predicates, constants, etc.) to encode some form of temporal reasoning, modeling heterogeneous non-Markovian relationships in such frameworks is not part of their design. Another popular branch of logic inspired by Prolog is Answer Set Programming (ASP) [43, 51], which employs stable model semantics [32] and SAT-inspired algorithms, supports non-monotonic reasoning, and can handle incomplete information, making it popular for knowledge representation, robotics, and bioinformatics [27]. However, classical ASP or its equivalent reasoning formalisms like Equilibrium logic [54] do not support temporal reasoning. Temporal Equilibrium Logic (TEL) [12] extends them by using modal temporal operators. However, inference in TEL is intractable [9], making it unsuitable for large problems, thus precluding it from the large-scale temporal reasoning experiments carried out in this work. Other notable temporal logics include Linear Temporal Logic [55] and Computation Tree Logic [16], but they are not designed for deductive reasoning; rather, they are primarily used for formal verification applications. We note that the semantic structures often used for the verification of such logics—e.g., Kripke structures [20] and Markov Decision Processes for probabilistic variants [17, 34]—inherently incorporate Markov assumptions, which we do not make in this formalism.

Similarly, in modern machine learning systems, despite numerous advances in reasoning about graphs and environments, the environment is typically modeled as a Markov Decision Process (MDP), where the next state and reward depend solely on the current state and action. However, in real-world applications, this is seldom the case [33] and, as a result, it is impossible to make an optimal decision based only on the current state [57]. Recent work on Reinforcement Learning (RL) [14] illustrates the errors introduced when applying standard RL algorithms like Q-learning to non-Markovian environments. RL algorithms are well-known for their substantial data and training time requirements. These demands become even more pronounced when dealing with environments characterized by non-Markovian dynamics. Most works have attempted to tackle this issue by making the reward non-markovian, but keeping the underlying dynamics of the system markovian [3, 30]. Meanwhile, Gupta et al. [33] approximates non-Markovian dynamics as a fractional dynamical system to reduce data demands of model-free RL. While these have shown promise, and despite the consensus for the need to incorporate non-Markovian dynamics in ML systems, significant progress has remained elusive due to prohibitive data and time requirements. An important direction for future work would be to use algorithms designed to learn non-Markovian policies on non-Markovian structures.

Classical logic programming approaches following MDPs cannot represent non-Markovian time relationships. The idea of allowing non-Markovian temporal dependencies – specifically, rules featuring heterogeneous time lags – was introduced in Annotated Probabilistic Temporal (APT) Logic [60], which extended concepts from Temporal Probabilistic Logic Programs (TPLP) [23] and was subsequently developed further in Probabilistic Doxastic Temporal (PDT) Logic [45] and by Doder et al. [26]. APT logic demonstrated that such rule structures enable expressive capabilities not achievable by Markov Decision Processes (MDPs). However, the deductive inference presented in these works is intractable. In this paper, we retain the expressive strengths of these non-Markovian constructs within our logical framework, while providing tractable semantics that facilitate efficient reasoning. We also investigate the use of non-Markovian policy

learning with annotated logic, a direction that, to our knowledge, has not been previously pursued in the literature, likely due to the intractable nature of APT logic inference.

Finally, in this paper we leverage a Skolemization procedure for efficiency that is possible due to our use of lower-lattice semantics. Skolemization [31] is a technique traditionally used for automatic theorem proving, and has been employed in predicate calculus for the substitution of existentially quantified variables with Skolem function applications for several decades [42, 49]. This process has subsequently been applied in the conversion of dynamic semantics into constants and functions [58], and the generation of normal forms for logical formulae [4]. More recently, it has been used for translation in Answer Set Programming [25]. However, we are not aware of any work that studies Skolemization with respect to annotated logic, including with respect to lower-lattice semantics. A major advantage of deductive or exact reasoning is that it ensures consistency and is often considered explainable [44]. However, black-box models, approximate or inexact reasoners, have become prevalent in real-world applications due to their flexibility and efficiency across domains involving diverse data structures and types [35, 68]. The proposed need-based grounding technique in LAT logic, based on Skolemization, significantly reduces the footprint of a program, allowing for a speed-up, which enables more extensive applications.

3 Technical Preliminaries

LAT logic extends Generalized Annotated Logic programs (GAPs) [38] by incorporating lower-lattice and temporal components. The lower lattice semantics were introduced in our prior work [61], which did not include temporal extensions, implementation, or many of the theoretical results in this paper. This framework employs GAPs with a lower lattice to model open-world scenarios, allowing atoms to be associated with a range of values beyond just “true” or “false”. The temporal extensions facilitate the representation of dynamic environments, while the underlying semantic structures and fixpoint approach enhance explainability in complex systems.

In subsequent subsections, we introduce the syntax and semantics of this logic. We also provide a running example that builds on the geospatial example introduced in Section 1.

3.1 Syntax

We consider a first-order logical language with finite sets C , \mathcal{P} , and \mathcal{V} of constant, predicate, and variable symbols, respectively. Following convention, we use uppercase letters for variables and lowercase for constants.

If a predicate symbol is directly applied to a list comprised of elements from the set of variables and constants ($\mathcal{V} \cup C$), it forms an atom.

DEFINITION 3.1 (ATOM). *If $E_1, \dots, E_n \in \mathcal{V} \cup C$, and $p \in \mathcal{P}$, then $p(E_1, \dots, E_n)$ is an atom.*

EXAMPLE 3.1 (ATOM). *Consider the geospatial example shown in Figure 2. From the instance at $t = 0$, we can have the following atoms:*

- (1) *Unary atoms: $agent(A)$, $location(L)$*
- (2) *Binary atoms: $at(A, L)$, $speed(A, S)$*

Here, A , L , and S are variables, and $agent$, $location$, at , and $speed$ are predicate symbols.

An atom, made up of only variables, is called a non-ground atom. If it contains both variables and constants, it is a partially ground atom. A fully instantiated atom, without any variables, is called a ground atom.

DEFINITION 3.2 (GROUND ATOM). *If $c_1, \dots, c_n \in C$ and $p \in \mathcal{P}$, then $p(c_1, \dots, c_n)$ is a ground atom.*

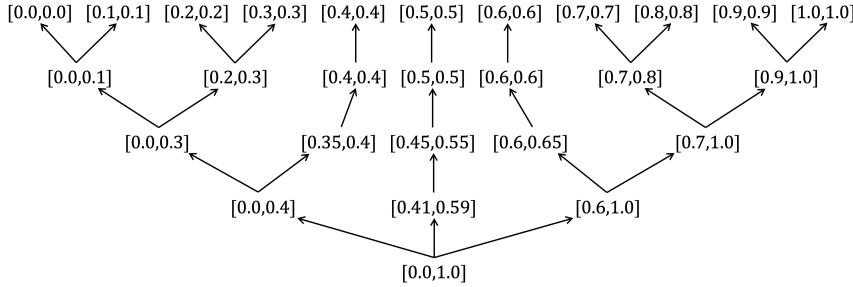


Fig. 3. Example of a lower semi-lattice structure where the elements are intervals in $[0, 1]$.

EXAMPLE 3.2 (GROUND ATOM). As mentioned previously in Section 1, the geospatial example domain has the following constants for typed variables A , L , and S :

\underline{A}	\underline{L}	\underline{S}
$footPatrol$	$locMid$	$slow$
$patrolCar$	$locLeft$	$fast$
	$locRight$	

Grounding the atoms in Example 3.1 using these constants, we get:

- (1) Unary ground atoms: $agent(footPatrol)$, $agent(patrolCar)$, $location(locMid)$, $location(locLeft)$, $location(locRight)$.
- (2) Binary ground atoms: $at(footPatrol, locMid)$, $at(patrolCar, locMid)$, $speed(footPatrol, slow)$, $speed(patrolCar, fast)$, . . .

Following [38], we define a lattice structure \mathcal{M} where elements consist of subsets of the real unit interval, where $[0, 1]$ (representing total uncertainty) is the lowest element of the lattice while the upper elements are all intervals $[\ell, u]$ where $\ell = u$; such elements include $[1, 1]$ (total truth) and $[0, 0]$ (total falsehood). Figure 3 illustrates an example of such a structure. One specific function we define is “ \neg ” for negation, which is used in the semantics of [38]. For a given $[\ell, u]$, $\neg([\ell, u]) = [1 - u, 1 - \ell]$. Note that we also use the symbol \neg as a connector in our first-order language (following the formalism of [38]). Now, we define annotations for atoms. We assume the existence of a set $AVar$ of variable symbols ranging over \mathcal{M} and a set \mathcal{F} of function symbols, each of which has an associated arity.

DEFINITION 3.3 (ANNOTATION).

- (1) Any member of $\mathcal{M} \cup AVar$ is an annotation.
- (2) If f is an n -ary function symbol over \mathcal{M} and m_1, \dots, m_n are annotations, then $f(m_1, \dots, m_n)$ is an annotation.

In annotated logic, atoms are associated with elements of the lattice structure, which enables open-world reasoning. We thus define, as in [61, 62], annotated atoms as $a : \mu$, where a is an atom and μ is an element of the lattice. An annotated literal is either an annotated atom or its negation. Functions and variables are also permitted in the annotations (see above references for further details).

EXAMPLE 3.3 (ANNOTATION). Continuing with our example and, as shown in Figure 2, an example of an annotated ground atom is $speed(footPatrol, slow) : [1, 1]$. Its equivalent negation is $\neg speed(footPatrol, slow) : [0, 0]$.

In order to extend the logic to make statements about time, and we follow the convention of [63] and define *temporal annotated facts* (TAFs).

DEFINITION 3.4 (TEMPORAL ANNOTATED FACTS (TAFs)). *For a literal a , an annotation μ , and a time point t , $a : (\mu, t)$ is a temporal annotated fact (TAF).*

We choose to use the notation $a : (\mu, t)$ for TAFs as it makes it quite distinct from annotated atoms without a temporal component (typically denoted by $a : \mu$). We note that, in a previous work [7], $a : \mu_t$ was used to denote TAFs; however, we have chosen to move away from that notation as subscripts are also used sometimes to differentiate between different literals, constants, components of a vector, etc.

EXAMPLE 3.4 (TEMPORAL ANNOTATED FACTS (TAFs)). *In our running example, we have three time points, $t = 0, 1, 2$. As the two agents move, their locations are represented by TAFs:*

$$\begin{aligned} t = 0 \quad & \text{at}(footPatrol, locMid) : ([1, 1], 0) \\ & \text{at}(patrolCar, locMid) : ([1, 1], 0) \\ t = 1 \quad & \text{at}(footPatrol, locMid) : ([0, 0], 1) \\ & \text{at}(patrolCar, locMid) : ([0, 0], 1) \\ & \text{at}(patrolCar, locLeft) : ([1, 1], 1) \\ t = 2 \quad & \text{at}(footPatrol, locMid) : ([0, 0], 2) \\ & \text{at}(patrolCar, locMid) : ([0, 0], 2) \\ & \text{at}(patrolCar, locLeft) : ([1, 1], 2) \\ & \text{at}(footPatrol, locRight) : ([1, 1], 2) \end{aligned}$$

Note how the “truth” about the location of “footPatrol” is uncertain at $t = 1$. However, open-world reasoning supports complete uncertainty, and hence if we were to ground any arbitrary point between the starting and final geo-locations of “footPatrol” at $t = 1$, say, $locMidRight$, then we can represent it with the TAF:

$$\text{at}(footPatrol, locMidRight) : ([0, 1], 1).$$

Annotated literals serve as the building blocks of a GAP rule. We propose the following definition of a modified version of the GAP rules defined in [61]:

DEFINITION 3.5 (GAP RULE). *If $\ell_0 : \mu_0, \ell_1 : \mu_1, \dots, \ell_m : \mu_m$ are annotated literals s.t. for all $i, j \in 1, \dots, m$, $\ell_i \neq \ell_j$, then:*

$$r \equiv \ell_0 : \mu_0 \xleftarrow[\Delta t]{} \ell_1 : \mu_1 \wedge \dots \wedge \ell_m : \mu_m, \text{ with } \Delta t \geq 0 \quad (1)$$

is called a GAP rule. We will use the notations $\text{head}(r)$, $\text{delay}(r)$, and $\text{body}(r)$ to denote ℓ_0 , Δt , and $\{\ell_1, \dots, \ell_m\}$, resp. When $m = 0$ ($\text{body}(r) = \emptyset$), the rule is called a fact. A GAP rule is ground iff there are no occurrences of variables in it.

Intuitively, Δt is the temporal gap between when the rule is fired and when its effects hold. If $\text{body}(r)$ is satisfied at time t , then the annotation of ℓ_0 changes to μ_0 at time $t + \Delta t$.

EXAMPLE 3.5 (GAP RULE). *Examples of GAP rules, and their English language equivalent, for the geospatial example are shown in Figure 1.*

We call rules with $\Delta t = 0$ immediate rules, which are applied as soon as the body is satisfied. Immediate rules relax the need for interdependent ground rules to be separated by not only applications of fixpoint operators but also actual

time points. We use $\Delta t = 0$ to approximate infinitesimal time intervals. We elaborate on how the implementation achieves this in Section 5.2.1.

A temporal logic program Π is a finite set of GAP rules that can be used to capture expert knowledge of an environment's dynamics or learned from data. Incorporating temporal constructs into literals and GAP rules enables explicit representation and reasoning over temporal dependencies, facilitating non-Markovian reasoning by capturing historical states and temporal relations beyond the current state.

3.2 Semantics

Annotated logic programs are associated with interpretations that map literal-time point pairs to annotations. Given a program, the intuition is that this structure (which, as shown below, can be produced as output of deductive inference) can directly describe changes in the environment or knowledge. Interpretations are symbolic, and hence support explainability based on the underlying logical language. We now provide a formal definition of interpretations and the associated satisfaction relationship.

DEFINITION 3.6 (INTERPRETATION). Let Π be a program, \mathcal{G} the set of all ground literals, and $T = t_1, \dots, t_{\max}$ a sequence of time points. An interpretation I is a mapping $\mathcal{G} \times T \rightarrow \mathcal{M}$ such that for all literals l , we have $I(l, t) = \neg(I(\neg l, t))$.

EXAMPLE 3.6 (INTERPRETATION). Consider the TAF $\text{at}(patrolCar, locMid) : ([1, 1], 0)$ from Example 3.4. The interpretation can be represented as: $I(\text{at}(patrolCar, locMid), 0) = [1, 1]$; its negation is: $I(\neg \text{at}(patrolCar, locMid), 0) = [0, 0]$.

The set \mathcal{I} of all interpretations can be partially ordered via the ordering: $I_1 \leq I_2$ iff for all ground literals $g \in \mathcal{G}$ and time t , $I_1(g, t) \sqsubseteq I_2(g, t)$. This set forms a complete lattice under the \leq ordering. An interpretation is said to satisfy an annotated ground literal at time t if its annotation is contained in the sub-lattice of its assigned value.

DEFINITION 3.7 (SATISFACTION). An interpretation I satisfies an annotated ground literal $g : \mu$ at time t , denoted $I \models_t g : \mu$, iff $\mu \sqsubseteq I(g, t)$.

We can then extend the definition of satisfaction from an annotated literal to a complete GAP rule as:

DEFINITION 3.8 (SATISFACTION OF GAP RULE). I satisfies the ground rule

$$r \equiv g_0 : \mu_0 \leftarrow_{\Delta t} g_1 : \mu_1 \wedge \dots \wedge g_m : \mu_m \quad (2)$$

denoted $I \models r$, iff for $t \leq t_{\max} - \Delta t$, where for all $g_i : \mu_i \in \text{body}(r)$, if $I \models_t g_i : \mu_i$ then $I \models_{t+\Delta t} \text{head}(r)$.

We say that I satisfies a non-ground literal or rule iff I satisfies all of its ground instances.

A program in our logical language is made up of both TAFs and rules.

DEFINITION 3.9 (GENERALIZED ANNOTATED PROGRAM (GAP)). GAP Π is a set of Temporal annotated facts (Π_{TAFs}) and GAP rules (Π_{Rules}). Given an Interpretation I and a program Π , $I \models \Pi$ iff

$$\forall x \in \Pi, I \models x$$

The next definition captures the central concept of consistency; intuitively, if a Generalized Annotated Program's rules and annotations yield stable, non-contradictory, logically sound outcomes during its evaluation, it is considered to be consistent.

DEFINITION 3.10 (CONSISTENCY). A GAP Π is consistent if there exists some I that satisfies all the rules in Π .

$$\boxed{\begin{aligned}\Pi_{\text{simple}} = \{ & b(X) : [1, 1] \xleftarrow[\Delta t=1]{} a(X) : [1, 1], \\ & c(X) : [1, 1] \xleftarrow[\Delta t=0]{} b(X) : [1, 1]\}\end{aligned}}$$

Fig. 4. A simple logic program Π_{simple} used to illustrate the application of fixpoint operator in Example 3.7.

A GAP is said to entail a TAF if it logically implies or derives the TAF from its rules and annotations under its semantic framework.

DEFINITION 3.11 (ENTAILMENT). *We say GAP Π entails TAF $a : (\mu, t)$, denoted $\Pi \models_{\text{ent}} a : (\mu, t)$, iff for every interpretation I s.t. $I \models \Pi$ we have that $I \models_t a : \mu$.*

A model of the GAP that assigns annotations to TAFs in a way that logically satisfies the program's rules and is minimal with respect to the lattice ordering of annotations is called the minimal model of the program. Minimal models thus represent the most “conservative” solutions (having the tightest annotation bounds) consistent with the program.

DEFINITION 3.12 (MINIMAL MODEL). *Given program Π , the minimal model of Π is an interpretation I s.t. $I \models \Pi$ and for all interpretation I' s.t. $I' \models \Pi$, we have that $I' \leq I$.*

As shown by [38], we can associate a fixpoint operator with any GAP Π that maps interpretations to interpretations. In [61, 62], the authors present a fixpoint operator for identifying the logical outcome of a logic program. Intuitively, this operator performs a simulation while recording changes. Under the assumption of consistency, this operator produces an exact result in polynomial time (see Theorems 3.2 and 3.4 in [60]), and our implementation provides practical speedups and consistency checking while maintaining these guarantees. We next define this operator formally:

DEFINITION 3.13 (FIXPOINT OPERATOR). *Let Π be a program and I an interpretation. The fixpoint operator Γ is a mapping defined as follows:*

$$\Gamma(I)(g_0, t) = \sup(\text{annoSet}_{\Pi, I}(g_0, t)),$$

where $\text{annoSet}_{\Pi, I}(g_0, t) = \{I(g_0, t)\} \cup \{\mu_0 \text{ such that for all ground rules } r \in \Pi, \text{ where } \text{head}(r) = g_0 : \mu_0, \text{ for all } g_i : \mu_i \in \text{body}(r), \text{ delay}(r) \leq t, \text{ and } I \models_{t-\text{delay}(r)} g_i : \mu_i\}$.

Note that the operator maps *all* timepoint-literal pairs to timepoint-literal pairs, essentially revising the entire sequence of timepoints at once. This contrasts with approaches such as Markov Decision Processes, which model transitions to a new state at each timepoint, and allows for direct modeling of non-Markovian dynamics.

DEFINITION 3.14 (ITERATIVE APPLICATIONS OF Γ). *Given natural number $i > 0$, interpretation I , and program Π , we define multiple applications of fixpoint operator Γ as follows: $\Gamma^i(I) = \Gamma(I)$ if $i = 1$, and $\Gamma^i(I) = \Gamma(\Gamma^{i-1}(I))$ otherwise.*

EXAMPLE 3.7 (FIXPOINT OPERATOR). *Consider the simple program Π_{simple} as shown in Figure 4. If the fixpoint operator is applied twice, notice how the interpretations change in Table 1. We assume a grounding of $X = x$ for this example. Further applications of the fixpoint applications do not lead to any further change in the set of interpretations. In such a scenario, we say that the fixpoint operation has converged.*

Finally, LAT logic additionally supports fuzzy logic [5, 37, 70] and other non-classical approaches by enabling arbitrary functions that can be used over real values or intervals of reals. This provides a key advantage to reasoning about constructs learned with neuro-symbolic approaches such as those explored in works such as [28, 56, 59, 61, 64].

Table 1. Evolution of interpretations as Γ is applied to Π_{simple} in Figure 4

t	I	$\Gamma(I)$	$\Gamma^2(I)$
1	$a(x):[1,1]$	$a(x):[1,1]$	$a(x):[1,1]$
2		$b(x):[1,1]$	$b(x):[1,1], c(x):[1,1]$
3	$a(x):[1,1]$	$a(x):[1,1]$	$a(x):[1,1]$
4		$b(x):[1,1]$	$b(x):[1,1], c(x):[1,1]$

4 Theoretical Analysis

This section provides a formal analysis of our framework, beginning with a detailed examination of the theoretical correctness of the fixpoint operator when applied to Generalized Annotated Programs (GAPs) extended with temporal constructs. Following this, we investigate the theoretical aspects of performance, emphasizing how the incorporation of a lower semi-lattice structure facilitates dynamic, on-demand creation of symbols via Skolemization, thereby enabling efficient reasoning in domains with potentially infinite constants. Together, these results establish both the soundness of our formalism and its practical scalability, laying a foundation for subsequent experimental evaluation.

4.1 Theoretical Results on Correctness¹

We first show the fundamental properties of the fixpoint operator when a GAP is consistent.

THEOREM 4.1. *If GAP Π is consistent, then:*

- (1) Γ is monotonic,
- (2) Γ has a least fixpoint $lfp(\Gamma)$, and
- (3) Π entails TAF $a : \mu$ iff $\mu \leq lfp(\Gamma)(a)$.

PROOF. (1 and 2) By creating an interpretation that maps atoms to annotations for time $t \in \{t_1, \dots, t_{max}\}$, the monotonicity of Γ is trivial even in the case where Π is inconsistent and it also has a least fixpoint by definition of the Γ operator.

(3) Suppose BWOC that Π entails $a : \mu$ and $\mu > lfp(\Gamma)(a)$. However, this would imply there is a series of logical constructs that allow us to derive $a : \mu$ at some time t , and this would trivially be reflected in the iterative applications of the Γ operator. Going the other way, BWOC if $\mu \leq lfp(\Gamma)(a)$ but Π does not entail $a : \mu$ would imply that there is no application of the constructs in Π that lead to the deductive conclusion of $a : \mu$ at any time t ; however this is again contradicted by the fact that Γ directly leverages the elements of Π . \square

We can also show that for GAPs, we can bound the number of applications of Γ until convergence. This means that the computation process is guaranteed to terminate after a finite number of steps, establishes that the semantics are well-defined, and allows for effective inconsistency checking. This is crucial for practical use because it ensures that inference based on these programs completes in a predictable and finite time.

THEOREM 4.2. *If GAP Π is consistent, then $lfp(\Gamma) \equiv \Gamma^x$ where $x = height(\mathcal{M}) * |\mathcal{A}| * t_{max}$.*

PROOF. We know, by the definition of Γ , for any $i \leq x$, that for all $a \in \mathcal{A}$, $\Gamma^{i(a)} \sqsubseteq \Gamma^{x(a)}$. Hence, we just need to consider the case where $i > x$ and $lfp(\Gamma) \equiv \Gamma^i$ and $lfp(\Gamma) \not\equiv \Gamma^x$. However, at each iteration the annotation of at least

¹A version of the results in Section 4.1 presented in an earlier conference paper from the authors in [61]; however, here we expand on them to include GAPs with temporal structures.

one literal must change. The bound on the number of changes in annotation is $\text{height}(\mathcal{M})$ (as the annotations must stay the same or increase monotonically, as Π is consistent by the statement). Hence, we have a contradiction. \square

We can also leverage the Γ operator to identify inconsistencies. Since it tracks how logical statements are derived step-by-step, it can flag when new inferences contradict previously derived facts, revealing inconsistencies in the program's knowledge base or rules. Furthermore, it can provide an explainable trace of where and why inconsistencies occur, enabling users to pinpoint the exact rules or data causing the conflict.

THEOREM 4.3. *GAP Π is inconsistent if and only if for value i , and ground atom a , there exist $\mu, \mu' \in \text{annoSet}_{\Pi, \Gamma^i}(a, t)$ where $\mu \not\subseteq \mu'$ and $\mu' \not\subseteq \mu$.*

PROOF. Claim 1: If there exist i, a such that the statement holds, then Π is inconsistent. Suppose, BWOC, that such an i, a pair exist and Π is consistent. We know, by the definition of Γ , that $\Gamma(\Gamma^i)$ must be an interpretation. However, as there is no element above both μ, μ' , Γ that $\Gamma(\Gamma^i)$ cannot be a valid interpretation.

Claim 2: If Π is inconsistent, then there exist i, a such that the statement holds. Suppose, BWOC, Π is inconsistent and there does not exist such an i, a pair. Then, this implies that for all $a \in \mathcal{A}$ there exists some i' where $\Gamma(\Gamma^{i'}) = \Gamma^{i'}$ which means for any $i'' > i'$ at time t , we have $\Gamma^{i'} = \Gamma^{i''}$. Therefore, by the definition of satisfaction, $\Gamma^{i'}$ must satisfy Π , which is a contradiction. \square

Note that this theorem does not depend on Theorem 4.1 (which has consistency as a requirement). Application of Γ can find an atom where the lower bound exceeds the upper bound (causing an inconsistency) if and only if Π is inconsistent, and this will always happen within a finite, polynomial number of applications of Γ . On the other hand, Γ is guaranteed to converge within a finite, polynomial number of applications if Π is consistent. As long as Γ has not converged, we may not make any conclusion about the consistency of Π .

These results rigorously establish the correctness, convergence, and inconsistency detection properties of the fixpoint operator for GAPs with temporal extensions, ensuring sound and tractable reasoning. Building on this solid foundation, the next subsection examines how the underlying lattice structure enables the dynamic creation of atoms and constants, offering significant performance improvements. This analysis sets the stage for practical improvements in scalability and efficiency when reasoning over complex or infinite domains.

4.2 Theoretical Results on Performance

The use of the lower lattice structure enables the creation of atoms and constants in an ad-hoc manner. In this section, we provide new formal arguments as to how such ad-hoc symbol creation can provide significant performance improvements. These results enable LAT logic to reason about temporal relationships in settings with potentially infinite constants without sacrificing performance.

We begin by providing an example for a non-temporal use case designed to provide an intuition on the use of a lower lattice for ad-hoc symbol creation. Figure 5 shows a logic program for reasoning about changes in a knowledge graph, and Figure 6 illustrates the knowledge graph before and after inference. This example is an excerpt from the YAGO03-10 [65] dataset, which we have used in our experiments in Section 6. The program has a single rule stating that “ X is a citizen of country Y if X is born in Z and Z is a city in Y ”. Figure 6 shows an example knowledge graph with three constants: *ben*, *miami*, and *usa*, which form two binary ground atoms: *bornIn(ben, miami)* and *cityIn(miami, usa)*, which are true. When the rule is grounded with $X = \text{ben}$, $Y = \text{miami}$, and $Z = \text{usa}$, the body is satisfied and the rule fires, creating a new binary ground atom *citizenOf(ben, usa)* as shown in the figure.

$$\Pi_{kg} = \{citizenOf(X, Y) : [1, 1] \leftarrow bornIn(X, Z) : [1, 1] \wedge cityIn(Z, Y) : [1, 1]\}$$

If X is born in city Z and Z is in country Y, then X is a citizen of Y.

Fig. 5. Example of a logic program Π_{kg} for the knowledge graph shown in Figure 6. English translation for each rule is provided.

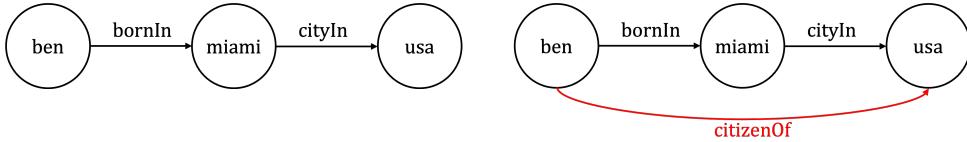


Fig. 6. Knowledge graph before (left) and after (right) inference. The newly created binary atom is shown in red.

Similarly, consider the geospatial example introduced in Section 1 with Π_{geo} , illustrated in Figure 2. When the first rule, which says, “*If fast-moving agent A is at L_1 and moves left, it will be at L_2 (left of L_1) after one time unit*” is fired, with the grounding $A = patrolCar$ and $L_1 = locMid$, a new constant $locLeft$ is created to the left of $locMid$ in space after one time point. In the same manner, firing of the second rule creates another constant $locRight$ after two time points. These create new unary ground atoms: $location(locLeft)$, $location(locRight)$, and binary ground atoms: $at(patrolCar, locLeft)$, $at(footPatrol, locRight)$, $left(locMid, locLeft)$, $right(locMid, locRight)$.

This illustrates the creation of a new constant in space dynamically and on demand, and consequently corresponding unary and binary atoms are grounded. Note the creation of a binary ground atom, and that *all* binary ground atoms in the language are implicitly assumed to exist even if not explicitly represented as facts. This is because of our use of a lower-lattice structure (i.e., all atoms are originally assigned a truth value at the bottom of the lattice). As uncertainty regarding the truth of atoms is reduced with the progression of inference (which moves up the lattice), *we only need to represent in memory those atoms that are not assigned a truth value associated with the bottom element of the lattice*.

In our open world formalism, anything that is not known to be true or false is uncertain—in practice, this means that everything that is uncertain (which could make up a vast portion of a practical KB) does not need to be allocated memory. In previous versions of our implementation, as well as other established software, the majority of symbols need to be grounded as the first step before carrying out inference. This limitation restricts the software to merely updating truth values or annotations to existing ground atoms based on available rules and facts at runtime. The increased memory consumption and computational overhead due to these factors are compounded due to the inherent sparsity of real-world knowledge bases and datasets. To mitigate these limitations, we introduce a Skolemization feature that allows our implementation to ground symbols only when certain rules are fired. This innovation eliminates the need for a complete grounding of symbols as a first step, allowing dynamic addition of new constants and ground atoms during inference. Consequently, we can now operate on incomplete knowledge bases while substantially reducing memory usage and running time. Empirical evidence supporting these improvements is presented in Section 6.

We now examine the theoretical impact of the Skolemization feature; its empirical impact will be studied later. In the following, we will refer to constant *types* as a way to split the set of constants C for modeling purposes to best represent the different properties (such as attributes in a graph) of the objects being modeled. Similarly, we can divide the set of variables V and predicates P into multiple subsets based on the domain being modeled. Finally, we assume a standard vector representation for each of these elements, and thus refer to their *components* in our analyses. The first result establishes an upper bound on the number of possible ground atoms:

Table 2. Notation used in Theorem 4.4.

g_i	Set of ground atoms after i applications of the fixpoint operator Γ .
g_0	Initial set of ground atoms.
P	Set of all possible predicates in the domain.
$g_i(p)$	Subset of g_i having predicate p , where $p \in P$.
$\text{pred}(\text{head}(r))$	Predicate in the atom in the head of rule r ; $\text{pred}(\text{head}(r)) \in P$.
$\text{pred}(\text{body}(r), j)$	Predicate in the j^{th} clause in body of rule r ; $\text{pred}(\text{body}(r), j) \in P$.

PROPOSITION 4.1. Consider an environment with t types of constants c_1, c_2, \dots, c_t . If each constant of type c_i has m components $c_{i,1}, c_{i,2}, \dots, c_{i,m}$ and the j^{th} component $c_{i,j}$ can take $n_{i,j}$ values, then the maximum possible number of constants is:

$$\sum_{i=1}^t \prod_{j=1}^m n_{i,j} \quad (3)$$

If the number of attributes for constant of type c_i is a_i , then the maximum possible number of ground atoms is:

$$\sum_{i=1}^t a_i \prod_{j=1}^m n_{i,j} \quad (4)$$

This result gives us two corollaries establishing bounds on the space required to store these atoms.

COROLLARY 4.1. Suppose that a constant is grounded in a vector with m components, and each component can be represented using b bits. Then, the maximum possible number of constants is 2^{bm} . Additionally, the amount of memory required to store these constants is $2^{bm} * (bm)$ bits.

COROLLARY 4.2. Let n_{hv} denote the upper bound on predicate arity, and $|P|$ denote the number of distinct predicates, where P is the set of all predicates. Then, the maximum number of ground atoms formed from 2^{bm} constants is $|P| * ((2^b)^m)^{n_{hv}}$. Additionally, if each ground atom takes b_a bits of memory, the amount of memory required to store all ground atoms is $|P| * ((2^b)^m)^{n_{hv}} * b_a$ bits.

Using these results, we now arrive at our main result regarding performance improvements; to improve readability, we summarize the notation used in Table 2.

THEOREM 4.4. Let Π_{Rules} be a GAP. The number of new ground atoms produced after the i^{th} application of the fixpoint operator cannot exceed:

$$\sum_{r \in \Pi_{\text{Rules}}} \prod_j |g_{i-1}(\text{pred}(\text{body}(r), j))| \quad (5)$$

A proof sketch is provided here. The full proof can be found in Appendix A.

PROOF SKETCH. Let $P_\Pi \subseteq P$ be the set of predicates containing only predicates present in the head of at least one rule in Π_{Rules} .

$$|g_i| = \sum_{p \in P_\Pi} |g_i(p)| + \sum_{p \notin P_\Pi} |g_i(p)|$$

As application of fixpoint operators can only create or modify atoms having a predicate in the head of a rule,

$$|g_i| = \sum_{p \in P_\Pi} |g_i(p)| + \sum_{p \notin P_\Pi} |g_0(p)| \quad (6)$$

Let $\Gamma_r(g)$ denote the set of ground atoms produced when a single fixpoint operator is applied to a single rule r with the set of ground atoms g .

$$\begin{aligned} |g_i(p)| &= |g_{i-1}(p) \cup \bigcup_{r \in \Pi_{rules} \wedge pred(head(r))=p} \Gamma_r(g_{i-1})| \\ &= |g_{i-1}(p)| + newF_{p,i} \times uniqueF_{p,i} \times \sum_{r \in \Pi_{rules} \wedge pred(head(r))=p} |\Gamma_r(g_{i-1})| \end{aligned} \quad (7)$$

Here, $newF_{p,i} \in [0, 1]$ denotes the fraction of ground atoms produced, with predicate p and at the i^{th} Γ application, which did not exist after the $(i-1)^{th}$ application. Similarly, $uniqueF_{p,i} \in [0, 1]$ is the fraction of ground atoms produced across rules, with predicate p in the head, which are unique. We now try to estimate the last term in Equation 7

$$|\Gamma_r(g_{i-1})| = validF_{r,i} \times \prod_j |g_{i-1}(pred(body(r), j))| \quad (8)$$

Here, $validF_{r,i} \in [0, 1]$ denotes the fraction of valid groundings that leads to firing of non-ground rule r , within the cross-product of possible groundings for each body clause.

From Equations 7 and 8, and considering the maximum value (= 1) for all three fractions we get:

$$\Delta|g_i(p)| = |g_i(p)| - |g_{i-1}(p)| \leq \sum_{\substack{r \in \Pi_{rules} \\ pred(head(r))=p}} \prod_j |g_{i-1}(pred(body(r), j))| \quad (9)$$

Substituting Equation 9 into Equation 6 we obtain:

$$|g_i| \leq \sum_{p \in P_\Pi} |g_{i-1}(p)| + \sum_{p \in P_\Pi} \sum_{\substack{r \in \Pi_{rules} \\ pred(head(r))=p}} \prod_j |g_{i-1}(pred(body(r), j))| + \sum_{p \notin P_\Pi} |g_0(p)| \quad (10)$$

We get $|g_{i-1}|$ by adding the first and last term on the right-hand side. The two summations can also be combined to give us our final expression:

$$\Delta|g_i| = |g_i| - |g_{i-1}| \leq \sum_{r \in \Pi_{rules}} \prod_j |g_{i-1}(pred(body(r), j))| \quad (11)$$

□

These results show the potential impact of reducing unnecessary groundings. In Section 6, we experimentally evaluate the impact of Skolemization and how it compares to this theoretical analysis.

5 Implementation

In this section, we elaborate on our implementation of LAT logic; specifically, we discuss our PyReason software and our efficient grounding process, and then provide details on using logic as a simulator, and how it can be interfaced readily with any reinforcement learning agent. Though earlier versions of the implementation were introduced in [1, 50], since then there have been multiple improvements. PyReason offers a comprehensive and flexible framework for reasoning based on generalized annotated logic; it supports various extensions, including temporal, graphical, and uncertainty-related features, which allow to capture a wide range of logics, such as fuzzy, real-valued, interval, and temporal logics. The open source codebase, tutorials, application case studies, and various other materials can be found at <https://pyreason.syracuse.edu>.

5.1 Knowledge Graphs as Data Structures Supporting Efficient Implementations

Built on modern Python, PyReason is specifically designed to handle graph-based data structures efficiently, and to support scalable yet accurate reasoning. We allow graphical input via the convenient *Graphml* format, making it compatible with data exported from popular graph databases like Neo4j and GraphML. The Python library Networkx is used to load and interact with the graph data.

A knowledge graph can be modeled using first-order predicate logic, where entities correspond to constants, relations correspond to predicates, and facts (usually represented as $(entity_1, relation, entity_2)$ triples in popular datasets) correspond to ground predicates (predicates with only constants as arguments). Constants correspond to nodes in the graph, while edges correspond to pairs of constants. Our approach, consistent with related literature [28, 36], restricts predicate arity to 1 or 2.

Algorithm 1 illustrates the structure of our implementation. Our software stores interpretations in a nested dictionary. Each constant forms a node, and edges correspond to pairs of constants that appear together in at least one grounded atom. For computational efficiency and ease of use, our software allows specification of a range of time-points $T = t_1, t_2, \dots$ instead of a single time-point t , for which an interpretation I remains valid. To reduce memory requirements, only one set of interpretations (current) is stored at any point in time. Once the inference engine is initialized, the program enters the “Main loop”, which includes all the necessary subroutines to cycle through each point in time. The fixpoint operator (Γ) is applied iteratively until convergence at each time point, and after every application, subroutines for logical consistency checking, followed by inconsistency resolution (if required), are run. Our optimized rule grounding process, key to efficient application of Γ is detailed in Section 5.2. In addition, the *persistent_flag* shown in Algorithm 1 provides users with the option of preserving annotations from inferences made at the previous time step (when the flag is set) or resetting them to the bottom element of the lattice, $[0, 1]$, (when the flag is reset).

The core of PyReason is its rule-based reasoning, which enables handling uncertainty, open-world novelty, non-ground rules, quantification, and other diverse requirements seamlessly. The system remains agnostic to the selection of t-norm², providing flexibility in using different logical connectives. Our description of the world as a knowledge graph (KG) adds support to applications where a policy must be learned via reasoning over context-related KGs such as [67]. Additionally, recent progress in developing Knowledge Graphs for probabilistic reasoning, as demonstrated by studies such as [11, 29, 66], highlights the potential role of our framework in a wide range of practical applications.

5.1.1 Special Case: Static Ground Atoms. In analyzing real-world use cases, we identified domain- or application-specific facts that remain constant over time. For instance, in the geospatial example introduced in Section 1, an agent’s top speed is an inherent property unaffected by time or inference. To model such cases in our logic, we introduce an optional *Static* flag. If a ground atom is declared with *Static* = *True*, its annotation remains fixed at the specified value across all time points. Algorithm 1 illustrates how the *Static* flag is incorporated into the data structures and checked before annotation updates. Ground atoms can be set to *static* through rules as well, preventing other rules from being able to update the same ground atom later.

5.2 Rule Grounding and Use of Skolemization

Many implementations of exact reasoning in different computational logic-based approaches struggle with the inherent complexity of the variable grounding process. The rule grounding process in LAT logic leverages a novel form of Skolemization enabled by the use of a lower lattice structure for annotations, which significantly reduces the creation

²A t-norm is a kind of binary operation that is typically used in fuzzy logic; they can be interpreted as a generalization of conjunction in classical logic.
Manuscript submitted to ACM

Algorithm 1 Open-World Annotated Temporal Logic implemented in PyReason**Data Structures**

- 1: Nested Dictionary $I = [Node/Edge, [Predicate, [Lower, Upper, Static]]]$ to store current interpretations only.
- 2: List $L = [(Node/Edge, Predicate, Lower, Upper, Static, at_t)]$ to store facts and inferences, before it is used to update the dictionary.
- 3: List $IPL = [(Predicate_1, Predicate_2)]$ containing pairs of predicates that cannot hold simultaneously (the bounds must be pairwise complementary). In the propositional case, if one of the predicates is *true*, the other must be *false*. We call this “inconsistent predicate list (IPL)”.
- 4: List $E = [(Node/Edge, Predicate)]$ containing a list of predicates that becomes inconsistent in the course of program execution.

Initialization

- 5: Initialize I as follows:
 - For each nodes/edges, use *type_checking* to initialize valid predicates only.
 - All bounds are initialized to $[0,1]$.
- 6: $L \leftarrow []$
Facts (including initial interpretations) are then copied into L
- 7: $t \leftarrow 0$
- 8: $E \leftarrow []$
- 9: Input: Number of diffusion time-steps T , Set of rules R

Main loop

- 10: **while** $t \leq T$ **do**
- 11: **for** i in I , if (persistent_flag == false) **do**
- 12: reset bounds to $[0,1]$ ▷ Annotations returned to bottom of the lattice.
- 13: **end for**
- 14: **update_req** $\leftarrow 0$
- 15: **for** l in L , where ($l(at_t) == t$) **do**
- 16: **if** check_consistency($l \in L, l \in I$) **then**
- 17: **update_req** += update_interp($l \in L, l \in I$)
- 18: **else**
- 19: **resolve_inconsistency**($l \in I$)
- 20: **if** $(l, l') \in IPL, \forall l'$ **then**
- 21: **resolve_inconsistency**($l' \in I$)
- 22: **end if**
- 23: **end if**
- 24: **end for**
- 25: **if** $update_req$ **then** ▷ Grounding process elaborated in Algorithm 2
- 26: Apply fixpoint operator(Γ) once.
- 27: **for** each resulting interpretation **do**
- 28: **if** *Static* is *false* in I **then**
- 29: Add to L
- 30: **end if**
- 31: **end for**
- 32: Go to line 14. ▷ Check if another Γ application is needed.
- 33: **else**
- 34: $t \leftarrow t + 1$.
- 35: **end if**
- 36: **end while**

of new ground atoms. This approach not only pursues tractable reasoning by bounding the size of groundings, but also facilitates scalable inference in complex temporal and non-Markovian environments. First, we optimize the grounding process through efficient constant search using a predicate hash map, and by using CPU parallelization methods detailed in Section 5.5. Then, by efficiently managing the introduction of new constants during grounding, Skolemization enhances both speed and memory usage, making it a key factor in the practical applicability of LAT logic for large-scale logic programs.

Algorithm 2 details the variable grounding process for a single non-ground rule in our implementation. After initializing lists for possible groundings, the variable dependency graph captures the inter-dependency between variables that co-occur in different clauses. This proves to be especially useful later in the process to rapidly trim down the combinations of possible groundings. This, in turn, significantly reduces the grounding search space by reducing the branching factor. First, we loop through all the body clauses to generate all possible candidate combinations of constants that simultaneously satisfy the complete set of annotations in the ground rule. After each clause is reviewed, the dependency graph is used to prune the list of candidates. Once this process is complete, if we have one or more satisfiable groundings, we apply the annotation function to compute the bounds for the ground atom in the head of the fired rule. We then check if the ground atom in the head of every ground rule fired exists in the graph and, if they do not, then the new constants are created at runtime. We then go back to the “Main loop” in Algorithm 1, which subsequently updates the interpretations.

5.2.1 Special Case: Immediate Rules. In section 3.1, we introduced immediate rules. They are rules with no delay, which are applied at the same time point at which the body of the rule is satisfied. Immediate rules make the program search for new applicable rules whose clauses might now be satisfied because of the immediate rule. Practically, rules with $\Delta t = 0$, are rules with infinitesimally small delay. This allows cascading of several rules, which are all fired within the same time point. A simple example was shown in Example 3.7. A practical use case can be found in our experiment in Section 6.3.2, when the shooting action is brought into the picture because multiple events are occurring within a single time point, but they’re all interconnected. We note that this is possible without any extensions to annotated logic as the temporal extensions we use (based on [1, 60, 62]) have no requirement that two time units be uniformly separated in actual time.

5.3 Logic as a Simulator for Reinforcement Learning

Deep Reinforcement Learning (RL) algorithms typically require a simulator to learn an agent policy. However, traditional simulators have several drawbacks, such as speed and data efficiency, as well as lack of explainability, modularity, and extensibility without retraining. We introduce PyReason-gym, an OpenAI Gymnasium wrapper that allows easy interfacing with a grid world that uses PyReason as the simulation and dynamics engine. In our experiment in Section 6.3, we show the applicability of our approach in a practical RL setting and compare it to two well-established simulation environments. By integrating PyReason as the underlying engine, the grid world dynamics and agent interactions are governed by logical rules, enabling precise and interpretable state transitions. Crucially, the temporal extensions and lower-lattice annotation of LAT logic means PyReason-gym can naturally interact with a simulator with non-Markovian dynamics, where agent behavior depends on multiple previous time steps rather than just the current state. PyReason-gym also uses a core functionality of our implementation to support the generation of detailed, time-stamped event traces, facilitating explainability in agent behavior and environment interactions. We discuss how this is particularly useful and show an example of an explainable trace in Section 5.4. With efficient memory usage and

Algorithm 2 Grounding a Non-Ground Rule

Require: A rule $r \in \Pi$ with $\text{head}(r)$, $\text{body}(r) = \{c_1, \dots, c_m\}$, thresholds $\Theta = (\Theta_1, \dots, \Theta_m)$, annotation function f_{ann} , interpretation \mathcal{I} , node set V , edge set E , and predicate-to-constant maps PredMap .

Ensure: Two lists of grounded instances: app_nodes if $\text{head}(r)$ is unary, app_edges if it is binary.

```

1: Extract variables  $\mathcal{V}$  and (if binary) head-edge pattern  $(X_h, Y_h)$ 
2: Initialize  $\text{groundings}[X] \leftarrow \emptyset$  for each  $X \in \mathcal{V}$ ,  $\text{groundings}_e[(X, Y)] \leftarrow \emptyset$  for each  $(X, Y)$  in binary clauses
3: Build variable dependency graph  $D$  from all binary clauses
4: for  $i = 1$  to  $m$  do
5:   Let  $c_i$  be  $(X : p : [\ell, u])$  if unary, or  $(X, Y : p : [\ell, u])$  if binary
6:   if  $c_i$  is unary then
7:      $S \leftarrow \begin{cases} V \cap \text{PredMap}[p], & X \text{ unseen} \\ \text{groundings}[X], & \text{otherwise} \end{cases}$ 
8:      $Q \leftarrow \{v \in S \mid \mathcal{I}(v, p) \sqsubseteq [\ell, u]\}$ 
9:      $\text{groundings}[X] \leftarrow Q$ 
10:    Prune any  $\text{groundings}_e$  entries involving  $X$ 
11:    if  $|Q|$  fails  $\Theta_i$  then return empty lists
12:   else if  $c_i$  is binary then
13:     Determine candidate edges
        
$$S \leftarrow \begin{cases} E \cap \text{PredMap}[p], & X, Y \text{ both unseen} \\ \{(v, w) \mid v \in \text{groundings}[X], w \in \text{Nbr}(v)\}, & Y \text{ unseen} \\ \{(v, w) \mid w \in \text{groundings}[Y], v \in \text{Rnbr}(w)\}, & X \text{ unseen} \\ \text{groundings}_e[(X, Y)], & \text{otherwise} \end{cases}$$

14:      $Q \leftarrow \{(v, w) \in S \mid \mathcal{I}((v, w), p) \sqsubseteq [\ell, u]\}$ 
15:      $\text{groundings}_e[(X, Y)] \leftarrow Q$ 
16:     Update  $\text{groundings}[X] \leftarrow \{v \mid \exists w : (v, w) \in Q\}$ ,  $\text{groundings}[Y] \leftarrow \{w \mid \exists v : (v, w) \in Q\}$ 
17:     Propagate refinements along  $D$ 
18:     if  $|Q|$  fails  $\Theta_i$  then return empty lists
19:   end if
20:   Propagate any changed  $\text{groundings}$  via dependency-graph pruning
21: end for
22: if all clauses satisfied then
23:   Initialize  $\text{app\_nodes}$ ,  $\text{app\_edges}$ 
24:   for all each tuple of groundings for head-variables do
25:     Locally re-refine and re-check all  $\Theta_i$ 
26:     if satisfied then
27:       Assemble annotation inputs from each clause's matches
28:       Compute  $[\ell', u'] \leftarrow f_{\text{ann}}(\dots)$ 
29:       Add any new constants or edges to  $(V, E)$ 
30:       if  $\text{head}(r)$  unary then append to  $\text{app\_nodes}$ 
31:       else append to  $\text{app\_edges}$ 
32:       end if
33:     end if
34:   end for
35:   return  $\text{app\_nodes}$ ,  $\text{app\_edges}$ 
36: else
37:   return empty lists
38: end if
```

Table 3. Rule trace produced by the PyReason software when Example 3.7 was executed.

t	Γ	Constant Symbols	Predicate	Old Annotation	New Annotation	Rule fired
1	0	x	a	[0.0,1.0]	[1.0,1.0]	–
2	1	x	b	[0.0,1.0]	[1.0,1.0]	rule ₁
2	2	x	c	[0.0,1.0]	[1.0,1.0]	rule ₂
3	0	x	a	[0.0,1.0]	[1.0,1.0]	–
4	1	x	b	[0.0,1.0]	[1.0,1.0]	rule ₁
4	2	x	c	[0.0,1.0]	[1.0,1.0]	rule ₂

configurable settings, PyReason-gym offers a practical, scalable solution for reinforcement learning applications, which can work with most out-of-the-box RL algorithm packages widely available. We also provide PyReason-gym as a fully open-source codebase, enabling anyone to easily reproduce and adapt it for their own application domains. We refer the interested reader to the PyReason website³ for more details.

5.4 Explainability

In order to support interpretability and explainability, interpretations used in past computations can be obtained using *rule traces*, which retain the change history for each interpretation and the corresponding grounded logical rules that caused each change. Such rule traces pave the way towards these goals, as every inference can be traced back to the cascade of rules that led to it. This enables users and developers to understand the exact reasoning path, identify and diagnose unexpected behaviors, and validate the correctness of complex temporal dependencies. Additionally, because LAT logic supports non-Markovian dynamics, rule traces are essential for capturing how past states and delayed effects contribute to current inferences, making temporal reasoning transparent and interpretable.

As an illustration of this functionality, Table 3 shows the rule trace for Example 3.7 where the fixpoint operator was applied twice to the program Π_{simple} . Note that to denote TAFs, the “Rule fired” column is intentionally left blank.

5.5 Software- and Hardware-based Performance Improvements

One of the key strengths of PyReason is its speed and machine-level optimized fixpoint-based deduction approach. This ensures efficient and scalable reasoning capabilities, even when dealing with large graphs with over 30 million edges. As stated before, the variable grounding process is one of the primary bottlenecks of exact reasoning implementations. Grounding non-ground rules is inherently expensive: to instantiate a rule with k variables over a domain of size N , the engine must consider up to N^k candidate substitutions at each iteration, and even simple two-variable bodies can require checking on the order of N^2 ground pairs. In a knowledge base with $N = 10^5$ constants, a rule like

$$h(Y) \leftarrow p(X) \wedge q(X, Y)$$

would naïvely generate 10^{10} combinations per fixpoint step—clearly prohibitive in pure Python without additional heuristics. To accelerate the grounding process and to streamline the whole inference engine, we make several software optimizations as well as leverage multi-core CPU processors for hardware optimizations.

³<https://pyreason.syracuse.edu/>

5.5.1 Software Optimizations. We list some of the design choices that leverage the properties of LAT logic to optimize our approach.

- (1) *Uncertain Predicate Filtering*: For each predicate p , maintain:

$$\text{PredFiltered}_p = \{v \in V \mid I(v, p) \neq [0, 1]\},$$

i.e. only those constants whose current annotation for p is not completely uncertain. When grounding a clause $p(X) : [l, u]$, we intersect V with PredFiltered_p , often reducing the domain by orders of magnitude.

- (2) *Predicate Maps*: In addition to *PredFiltered*, we maintain for each predicate p a map:

$$\text{PredMap}_p = \{v \in V \mid \exists t : p(v) : \mu_t \in \mathcal{I}\},$$

and similarly for edges in E . When grounding $p(X)$ or $p(X, Y)$, we initialize the candidate set from PredMap_p rather than the entire V or E . This enforces an open-world pruning based on known TAFs and avoids scanning the full graph.

- (3) *Clause Order Optimization*: Within each rule, we reorder the body so that all unary clauses appear before binary clauses. Since $|V| \ll |E|$ in typical sparse graphs, grounding the unary literals first shrinks the candidate *groundings*[X] sets, dramatically reducing the cost of the subsequent binary clause groundings $q(X, Y)$.
- (4) *Early Threshold Checks*: Immediately after grounding the i^{th} body literal c_i and obtaining its candidate set Q_i , we verify its counting or percentage threshold Θ_i . Concretely, for a clause:

$$|\{Y \mid q(X, Y) : [l, u]\}| \geq k,$$

we compute $|Q_i|$ once—if $|Q_i| < k$, the entire rule cannot fire for this X , so we abort grounding the remaining clauses and move on. This saves the cost of further joins and refinements when a single clause already fails.

- (5) *Dependency-Graph Pruning*: We build a small dependency graph D over the rule’s logic variables; each binary clause $q(X, Y)$ adds connections $X - Y$. Whenever a clause refines *groundings*[X], we propagate that change along D to remove any now-invalid values from neighboring *groundings*[Y] sets (and vice versa). This reduces the search spaces for future clauses and is able to stop the grounding process early if previous clauses no longer have their thresholds satisfied.

5.5.2 Hardware Accelerations. Grounding and fixpoint iteration are governed by a few core compute kernels—rule-grounding loops, lattice-join updates, and annotation propagation—that repeatedly traverse large portions of the graph representation. In naïve Python these loops suffer both interpreter overhead and poor memory access patterns, rapidly becoming the bottleneck as the number of rules or graph size grows. To overcome this, we compile and cache all of these critical loops with Numba’s Python `@njit(parallel=True)` decorator, yielding LLVM-generated [40] native code.

Furthermore, each fixpoint iteration naturally decomposes into independent tasks—one per rule—since grounding and annotation for rule r_i only reads the “old” interpretation and writes its own proposed updates. We exploit this by replacing standard `for` loops over rules with Numba’s *prange* (parallel range function), which:

- (1) *Distributes rules across cores*: Each CPU thread compiles and executes a disjoint chunk of the rule list, grounding and computing annotation updates in parallel.
- (2) *Minimizes synchronization*: Threads accumulate their local updates in thread-local buffers. At the end of the iteration a single, lightweight reduction step merges each thread into the global interpretation with only $O(R)$ overhead (where R is the number of rules).

- (3) *Adaptively balances load*: Numba’s runtime uses dynamic scheduling to hand off new rules to idle threads, smoothing out any variability in grounding cost between simple and complex rules.

The combination of targeted software optimizations and efficient hardware-level parallelization significantly accelerates PyReason’s grounding and fixpoint computation processes. This integrated approach enables scalable, high-performance reasoning even on massive graphs that would otherwise be computationally prohibitive.

6 Experimental Evaluation

In this section, we present the results of our empirical evaluation, which complement the theoretical analysis from Section 4.2, and then go on to explore the computational benefits of leveraging Skolemization. We begin with an application in a geospatial domain, showcasing the creation of new constants and the resulting scalability benefits. Next, we explore Knowledge Graph completion tasks on popular datasets to illustrate the utility and scalability of our approach in knowledge extraction tasks. Finally, we extend our evaluation to Reinforcement Learning (RL) scenarios, where we demonstrate the scalability, portability, and explainability of our logic-based simulator in PyReason for complex game environments. These RL experiments also highlight the benefits of incorporating non-Markovian dynamics—which capture dependencies beyond the current state—and logic shielding—which provides safety constraints or policy guidance—to improve learning in challenging settings.

6.1 Creation of Constants in a Geospatial Application

This experimental setup demonstrates the efficacy of our Skolemization technique in dynamic geospatial environments, highlighting its computational and memory advantages. The experiment involves a series of geospatial areas (maps) defined over spaces with varying granularity, ranging from resolution 2 (two levels of nested quadrants equaling 16 points) to resolution 9. Effectively, we may visualize a fully grounded out map as a grid. These resolutions define the set of possible constants. The logic program in each experiment is designed to model a two-team game scenario, where agents navigate the map according to user-provided directions, constrained by grid space and agent type. Each team has two types of agents in this game, and each type has unique movement restrictions: (a) *Border agents*: Limited to edges of the area, with one step per time unit. (b) *Field agents*: Unrestricted movement, capable of two steps per time unit. Leveraging the logic’s capability of handling non-Markovian properties, two differing speeds are used, which is shown to be relevant in practical deployments. Each team’s agents begin at a corner, representing the “minimum” scenario for ground atoms in the Skolemization approach.

Leveraging PyReason as the inference engine, the experiment compares the two approaches. Key distinctions in initial graph construction include:

- (1) *Non-Skolemization*: All spatial points and immediate neighbor edges are defined as constant symbols (nodes), resulting in a graph size proportional to map resolution.
- (2) *Skolemization*: Initially grounds only points that are occupied by agents, plus their immediate neighbors, dynamically grounding new constants during reasoning based on movements.

The Skolemization graph size correlates with agent positions rather than map resolution, potentially offering significant computational advantages in sparse environments.

6.1.1 Reduction in Groundings with Skolemization: Creation of New Constants in a Geospatial Domain. In Section 4.2, we derived expressions for the maximum number of ground atoms with and without Skolemization (Eqs. 5 and 4, Manuscript submitted to ACM

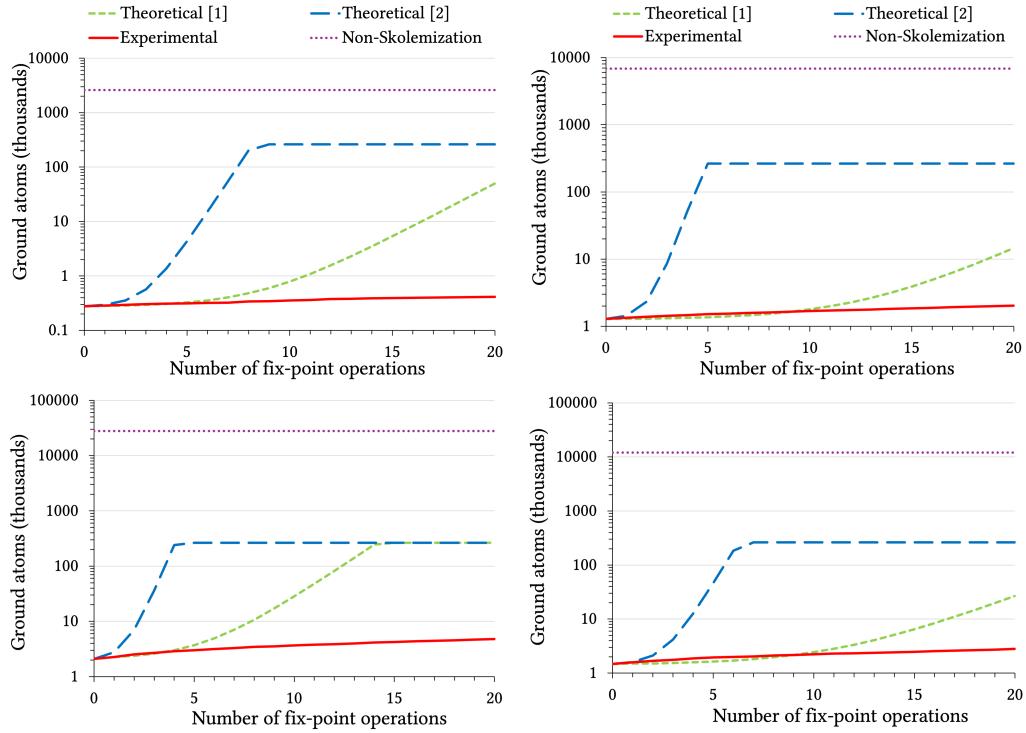


Fig. 7. (Clockwise from top-left) Comparison of #ground atoms with multiple applications of fixpoint operator (Π_4 , Π_{20} , Π_{40} , Π_{100}).

respectively). Now, we experimentally aim to study how these expressions compare with empirical observations in our aforementioned use cases.

We create four programs, one each for when each of the two teams has 2 (Π_4), 10 (Π_{20}), 20 (Π_{40}), and 50 (Π_{100}) agents. We can think of minimal model computation via the fixpoint operator on these programs as essentially simulating agent movements. In our experiments, we allow for twenty time steps (note that time steps are the amount of time we represent, and are not necessarily related to the number of fixpoint operations). The fully-grounded version of the graph contains 262,144 geographic constants (represented by nodes in the graph), and the largest graph (for 100 agents) is comprised of more than 27.7 million ground atoms. In comparison, the graph before simulation for the Skolemization approach only contains 2,084 ground atoms. Results of the simulations are shown in Figure 7 (note that the y -axes are log scale). We choose two parameter values for each setting to plot the theoretical bounds that closely mirror experimental values for the first reasoning step. We note that the theoretical bound is generally tight for lower numbers of inference steps (fixpoint operations), and the tightness of the bound varies based on the parameter. Further, we observe that, even when theoretical results converge after certain number of fixpoint operations, it is up to a few orders of magnitude below the number of ground atoms for the Non-Skolemization case. All theoretical values converged before 50 fixpoint applications when run to convergence.

6.1.2 Scalability: Scaling with Ground Atoms. In the aforementioned geospatial domain, we again make use of programs Π_4 , Π_{20} , Π_{40} , and Π_{100} ; for each case, we simulated agent movements for 100 actions while varying map size, thereby

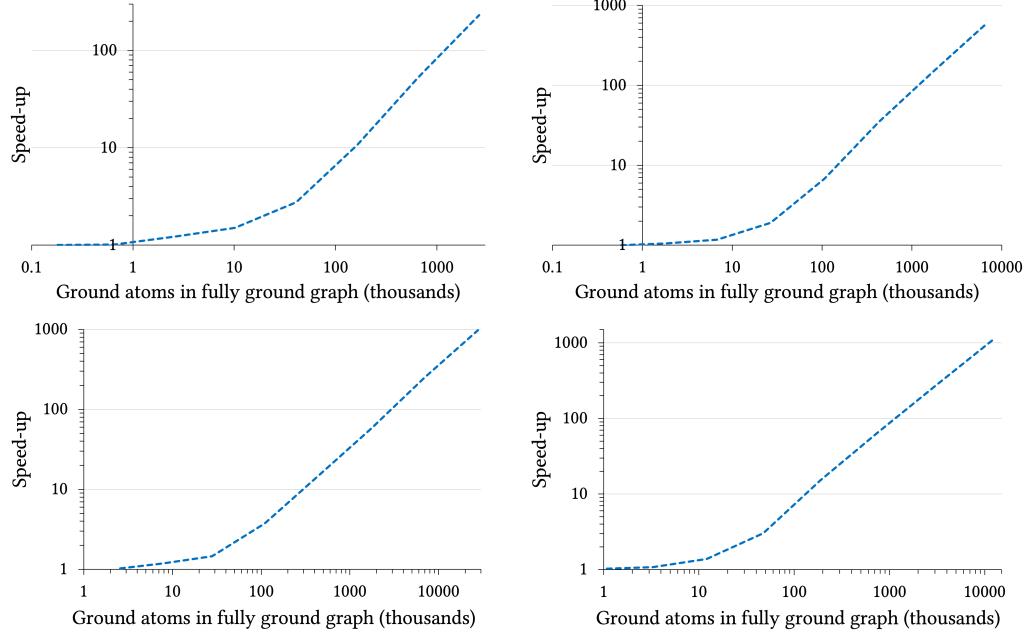


Fig. 8. (Clockwise from top-left) Speed-up vs Map size for Π_4 , Π_{20} , Π_{40} , Π_{100} .

varying the number of ground atoms in the fully-grounded graph (Non-Skolemization case). Figure 8 illustrates the observed speedup, defined as the ratio of running time for the non-Skolemization approach versus the Skolemization approach, across various experimental settings—note that both axes in these plots are log scale. A discernible pattern emerges wherein the magnitude of speedup increases substantially as the number of ground atoms escalates (corresponding to larger graph sizes). Moreover, the actual speedup demonstrates a positive correlation with the number of agents; this observation aligns with intuitive expectations, as a greater number of agents requires more groundings during the reasoning process. Figure 9 depicts the reduction in memory footprint achieved by our approach (again, both axes are log scale). Notably, we observe memory savings of up to 60GB for the largest graph with the highest number of agents.

6.2 Knowledge Graph completion with multi-step reasoning

We conducted Knowledge Graph (KG) completion experiments on four standard datasets: UMLS [46], YAGO03-10 [65], FB15k-237, and WN18RR [8]. To obtain extensive runtime and memory footprint data within a generous time limit, we created subsets of the latter three datasets using stratified sampling, preserving the underlying structure and relation types—Table 4 provides details of the datasets used. The Skolemization approach graph was limited to training set triplets, while the Non-Skolemization graph included all possible edge relations, resulting in a direct relationship between graph size and the number of constants in the training set for the Non-Skolemization approach. We used the publicly available AnyBURL rule learner [47] to generate rules, limiting the rule learning process to 20 minutes per dataset for consistency and efficiency. We filtered for rules with minimum 70% confidence, and obtained between a few and several thousand rules for different datasets.

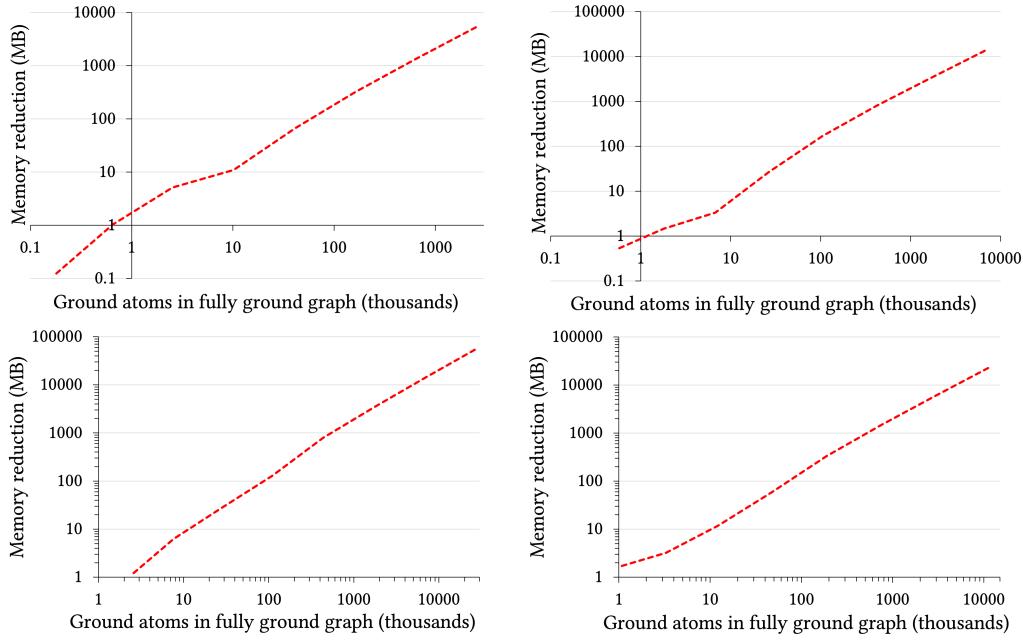


Fig. 9. (Clockwise from top-left) Memory reduction (in MB) vs Map size for Π_4 , Π_{20} , Π_{40} , Π_{100} .

Table 4. Knowledge graph datasets used (*subsets).

Dataset	Nodes	Edges	Unique Predicates
UMLS	135	5,216	46
FB15k-237*	945	1,108	237
YAGO03-10*	3,029	5,020	37
WN18RR*	8,809	10,007	11

KG completion was performed in PyReason using the created graph and generated rules, and evaluation results were computed using AnyBURL’s evaluation script. To assess the utility of multi-step reasoning for KG completion tasks we use the metrics hits@ k (the fraction of true predictions that appear in the top- k predictions), precision (ratio of true positives to the total number of predictions), and recall (ratio of true positives to the total number of relevant instances)—we include the latter two in order to better understand KG completion performance, particularly to observe the impact of multi-step reasoning.

6.2.1 Reduction in Groundings with Skolemization: Grounding in a Sparse Knowledge Graph. For each of the four generated knowledge graphs, we perform multi-step reasoning and observe how many new groundings are made after each fixpoint operation. To ensure the feasibility of our comprehensive experimental suite within reasonable time constraints, we employed the following combinations of program size (quantified by ground rules) and fixpoint operations for each dataset:

- UMLS: 10,000 rules, 20 fixpoints;

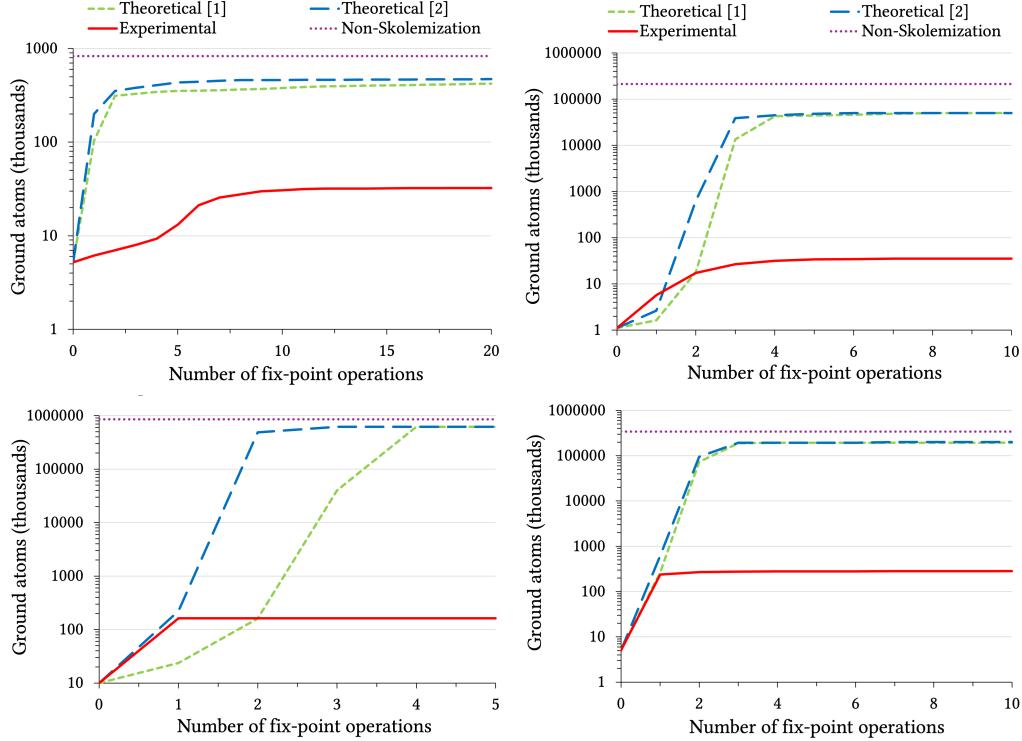


Fig. 10. (Clockwise from top-left) Comparison of number of ground atoms with multiple applications of fixpoint operator for UMLS, FB15k-237, YAGO03-10, and WN18RR.

- FB15k-237: 1,565 rules, 10 fixpoints;
- YAGO03-10: 1,000 rules, 10 fixpoints; and
- WN18RR: 100 rules, 5 fixpoints.

Results are plotted in Figure 10 (y axes on log scale), and show similar characteristics to those of the geospatial domain. The theoretical limit, while initially approximating experimental values at lower fixpoint operations, rapidly increases before stabilizing at a generous upper bound. We note that cases where the number of constants produced exceeds the theoretical bound (in lower inference steps) are due to the choice of parameters. Despite this, the upper bound consistently remains significantly below the number of ground atoms for the non-Skolemization approach. An intriguing observation emerged regarding the influence of unique predicates in the head of non-ground rules within a program. A higher diversity of predicates typically correlated with delayed convergence and a greater number of ground atoms at convergence. These findings support our hypothesis that practical applications require materializing only a small fraction of all possible groundings. The Skolemization approach thus facilitates efficient reasoning in logic programming by substantially reducing the number of required ground atoms.

6.2.2 Scalability: Scaling with Rules and Fixpoint Applications. To investigate the impact of program size and reasoning steps on running time and memory footprint, we selected the UMLS knowledge graph; this choice was based on its manageable file size and reasonable experimental time requirements. For context, while the fully-grounded UMLS graph

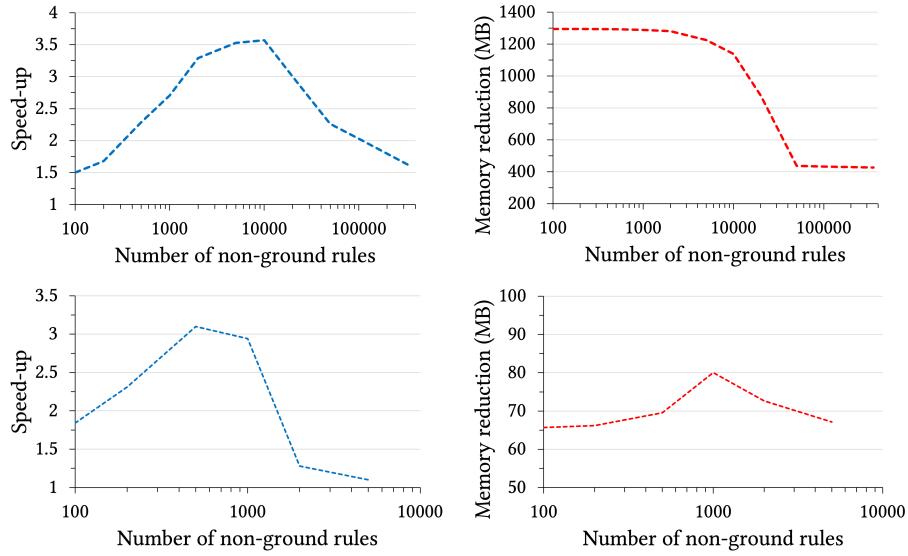


Fig. 11. Speed-up and Memory reduction vs Program size for 2 (top) and 5 (bottom) fixpoints on UMLS dataset.

requires 1.2GB of memory, our subgraphs of FB15k-237, YAGO03-10, and WN18RR required 32GB, 66GB, and over 100GB of disk space, respectively. Moreover, these larger graphs needed up to 1TB of memory for reasoning operations.

Figure 11 illustrates the speed-up and memory reduction (in MB) for 2 and 5 reasoning steps (x-axes on log scale). Maximum speedup was obtained between 1,000 and 10,000 rules, consistently outperforming the non-Skolemization approach. From two to five fixpoints, the peak efficiency shifted slightly towards a lower number of rules, corroborating our previous findings that increased fixpoint operations lead to more inferences and, consequently, increased running time, thus reducing speedup. With fewer rules, the growth of ground atoms is slower, resulting in faster running times. Similarly, memory efficiency decreases as the number of rules increases, due to the significant increase in inferences after each fixpoint. We imposed a 12-hour time limit for each experiment, consequently obtaining results for up to 5K non-ground rules for five fixpoints, compared to 355K for two fixpoints. Experiments were conducted on 128 cores of AMD EPYC 7413 with a maximum of 1TB allocated memory.

6.2.3 Multi-step Inference. Multi-step reasoning employs sequential applications of logical inference to build upon previously established knowledge after each fixpoint application, distinguishing it from multi-hop reasoning, which refers to the process of aggregating and connecting information across multiple pieces of evidence or sources. In our approach, we perform multi-step reasoning while relaxing any closed world assumption. We hypothesize that multi-step reasoning can help uncover deeper knowledge in a variety of scenarios, and we present some of our early findings in this direction. We conduct our experiments on subsets of all four datasets to test our hypotheses on graphs of significantly varying sizes, density, and with different number of rules. Table 5 illustrates that multi-step inference, involving just two fixpoint applications, consistently enhances performance across the Hits@k and Mean Reciprocal Rank (MRR) metrics, compared to single-step inference. Notably, FB15k-237 demonstrates improvements of 7.4%, 14.8%, and 13.7% on average for Hits@k. The MRR is also shown to always improve, with increases of over 25% for the WN18RR and FB15k-237 datasets. These findings indicate that multi-step inference could significantly enhance result retrieval capacity

Table 5. Performance metrics with single and multi-step inference

Dataset	Γ	Hits@k			MRR	# ground atoms	# rules	# nodes	# edges	# queries
		1	3	10						
WN18RR	1	0.043	0.065	0.092	0.058	725,666	500	8,809	10,007	6,268
	2	0.043	0.087	0.127	0.07	763,636				
FB15k-237	1	0.101	0.121	0.202	0.1248	1,676	400	945	1,108	40,932
	2	0.136	0.161	0.237	0.162	4,131				
YAGO03-10	1	0.357	0.429	0.429	0.393	107,014	500	3,029	5,020	10,000
	2	0.357	0.464	0.464	0.413	115,078				
UMLS	1	0.054	0.093	0.099	0.073	186	200	135	5,216	1,322
	2	0.055	0.097	0.104	0.076	267				

with multiple applications of the fixpoint operator, representing preliminary evidence of the approach’s potential that warrants further investigation.

6.3 Logic as a Simulator for RL Applications

In this section, we benchmark our approach against two popular simulators. We begin by introducing the simulators, then we outline the two game scenarios we use in our experiments, analyze the limitations of Markov assumptions, and discuss the RL training methodology adopted. Then, we present experimental results comparing our approach’s scalability in Starcraft II and AFSIM, along with its ability to learn policies in PyReason and port them to other simulators. We then explore whether incorporating non-Markovian dynamics in the simulation can improve RL algorithms’ ability to learn effective policies for complex games. Additionally, we demonstrate the explainability of our approach using rule traces, highlighting its potential in reward shaping during training.

6.3.1 Benchmarks: Popular Simulators. In order to position PyReason as an appropriate simulator, we first compare it to two established simulators in the field:

- (1) *Starcraft II* (SC2) is a popular real-time strategy (RTS) video game developed by Blizzard Entertainment, and has a competitive multiplayer aspect that involves managing resources, building armies, and engaging in tactical battles. Due to its complex gameplay and emphasis on strategic decision-making, it has been considered as a potential tool for military simulations. We extended Deepmind’s PySC2 [69] to use the Starcraft II environment in our experiments⁴.
- (2) *Advanced Framework for Simulation, Integration, and Modeling software* (AFSIM) [18] is a powerful simulation tool used by the United States Department of Defense (DoD) for various purposes, including training, analysis, experimentation, and mission planning. AFSIM is developed by the Air Force Research Laboratory (AFRL) and is used primarily by the United States Air Force (USAF) as well as other branches of the military and defense organizations. AFSIM is a high-fidelity modeling and simulation software designed to provide realistic representations of aerial warfare scenarios and environments. It enables the USAF to assess and analyze the performance of various systems, strategies, and tactics in simulated combat situations.

To compare PyReason with SC2 and AFSIM, we design the scenarios and game dynamics in all three simulators.

⁴Extensions to PySC2: <https://github.com/lab-v2/pysc2-labv2>

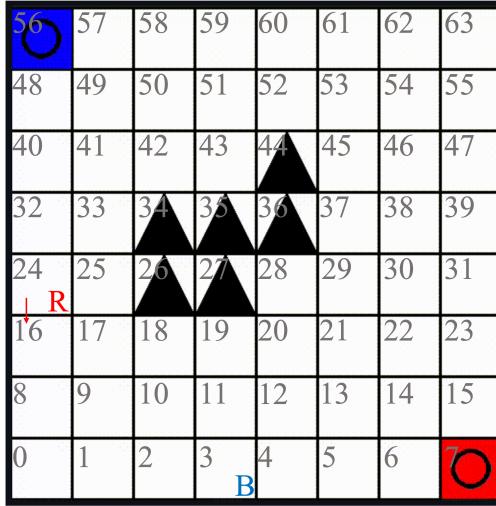


Fig. 12. Grid map for the scenario. Red (bottom-right) and Blue (top-left) squares are fixed base locations for each team. All agents start at their respective base locations. Obstacles (mountains) are shown with black triangles. Bottom left quadrant of the grid map is marked with indices to aid the understanding of the explainable trace in Table 8.

6.3.2 Game Setup. We design a simple grid world war game as shown in Figure 12. The basic scenario has two teams (red and blue) of one agent each. Each team has a base, and there are also a few obstacles (shown as mountains) in the environment that are impenetrable and impassable. For this base scenario, the objective of the game is to capture (reach) the rival base before the enemy can do the same. The red team follows our learned RL policy (the agent(s)), whereas the blue team follows a pre-defined base policy (the opponent(s)) described later in this section. Later on we build upon this basic scenario by adding more agents and then extending the action and observation spaces.

6.3.3 Scalability. Allowing the agents to take random actions in the grid world, we compare the scaling capability of our software against other simulators by comparing the running time and memory footprint over a large number of actions for different number of agents per team. The experiments were performed on an AWS EC2 container with 96 vCPUs (48 cores) and 384GB memory.

Figure 13 show the scaling capability of the different simulators tested. We note that, among the two established simulation environments, AFSIM generally performed better with 5 agents per team; with 20 agents per team, AFSIM is overtaken by SC2 as the actions per agent increase. This is expected, as AFSIM is designed as a high-fidelity simulation environment, so we can expect greater computational cost with more complex situations. PyReason consistently outperformed SC2, achieving anywhere from a one to nearly three orders of magnitude improvement. Though PyReason performs comparably to AFSIM for lower numbers of actions per agent (which are arguably the least important in practice), it also achieved comparable multiple order-of-magnitude improvements in terms of running time as the number of actions per agent increased. This suggests that PyReason will scale to large environments where the traditional use of simulators would otherwise prohibit model training.

Additionally, we examined memory consumption (Figure 13). PyReason uses considerably less memory compared with SC2 over all configurations while having sub-linear ($R^2 = .84$) growth with action and agent space. AFSIM's

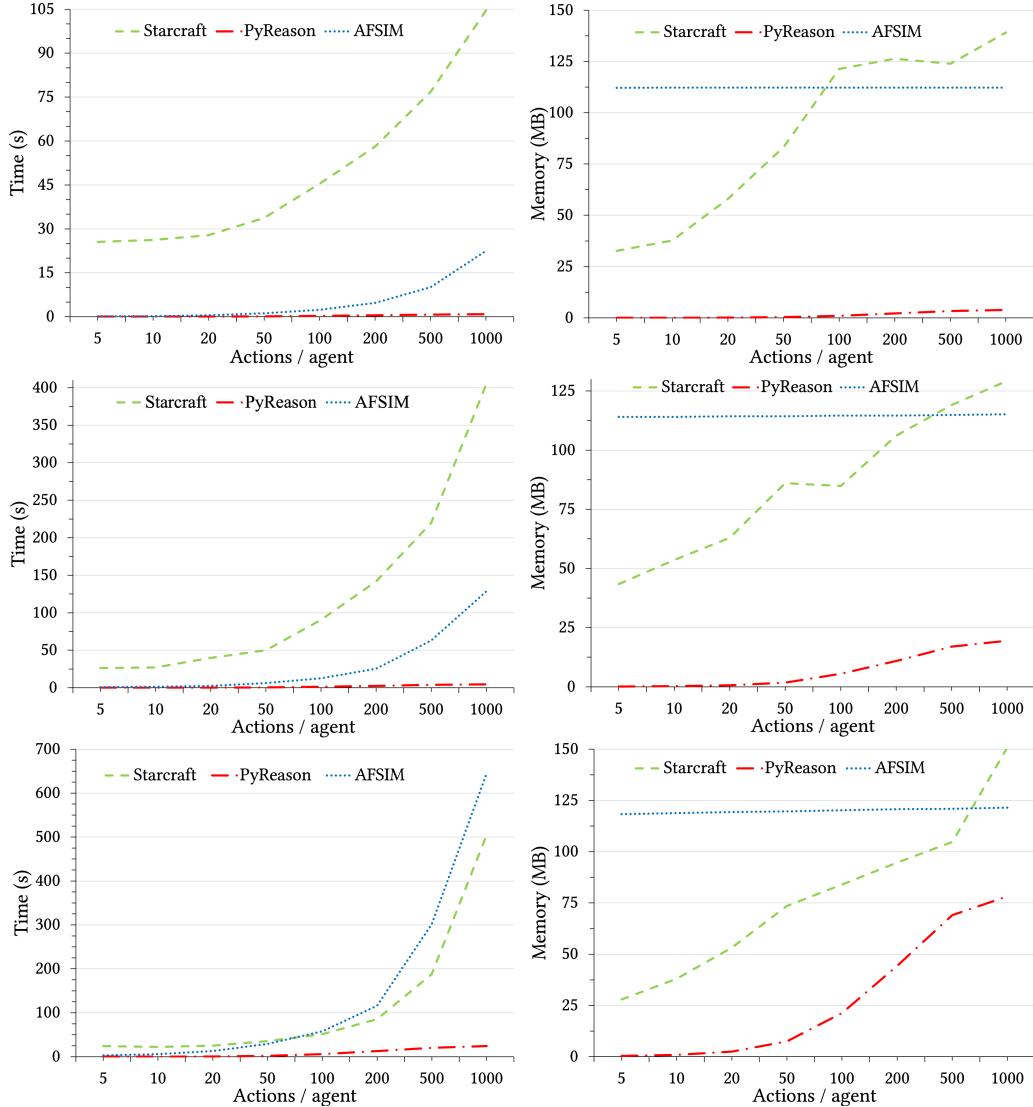


Fig. 13. Runtime (left) and memory footprint (right) comparison when 1 (top), 5 (middle), 20 (bottom) agents/team take random actions in three simulation environments.

strength as a large-scale military simulator is shown here with little effect on memory consumption with change in agents or actions; however, it has a large base memory cost that was still significantly higher than that of PyReason for the largest case considered (40,000 actions in total).

6.3.4 Portability. Next, we wanted to test whether a Reinforcement Learning (RL) agent trained in PyReason (PR) can provide performance comparable to AFSIM (AFS) and PySC2 (SC2). For this, we considered two cases: single agent and multi (five) agents per team. At certain intervals during the training process, policies were extracted and were used to

Table 6. Performance metrics when PyReason trained policies were used to play the game on different simulators for single and multi (5) agent scenarios—numbers in parentheses specify differences with respect to PyReason.

#	Epochs	Avg. Reward			Win %		
		PR	SC2	AFS	PR	SC2	AFS
1	400K	-209.87 (-0.13%)	-210.15 (-6.09%)	-222.65 (0)	0.0 (0)	0.0 (0)	0.0 (0)
	544K	162.51 (+1.93%)	165.64 (+3.40%)	168.04 (-0.2)	43.0 (+1)	42.8 (+1)	44.0 (+1)
	760K	482.50 (+0.93%)	487.00 (-1.87%)	473.50 (+2.4)	97.6 (+2.4)	100.0 (+2.4)	100.0 (+2.4)
5	112K	-913.27 (-8.06%)	-986.88 (+3.63%)	-880.16 (0)	0.0 (0)	0.0 (0)	0.0 (0)
	352K	-5166.99 (-7.38%)	-5548.18 (-1.21%)	-5229.43 (+0.2)	1.6 (-1.6)	1.8 (-1.6)	0.0 (-1.6)
	1536K	1899.71 (-2.09%)	1860.05 (-7.07%)	1765.43 (-0.6)	79.4 (-0.6)	78.8 (-0.6)	79.0 (-0.4)

play the base scenario described earlier 500 times in each of the three simulators (PyReason, AFSIM, and PySC2) and the outcomes were compared.

When policies learned in PyReason played the base scenario, comparable numbers were observed for all three simulators, as shown in Table 6—variance can be attributed to inherent randomness in learned policies). These results suggest that the approach is generalizable, as an agent trained in PyReason can be ported to various simulation environments and achieve comparable reward and win percentage.

6.3.5 Extending the Action Space with Shooting in PyReason. Some simulations (e.g., Starcraft II) do not separate movement and shooting (i.e., the agent always shoots when in line of sight with an enemy). This, however, is clearly undesirable in any military simulator looking to emulate real battlefield scenarios. Strategies are often pragmatic, with shooting typically limited and highly tactical, given that practical issues such as limited ammunition and avoiding exposure are important considerations here. Hence, we build upon the basic scenario by integrating shooting into PyReason, independent from movement actions, allowing RL agents to learn varied and in-depth strategies, and in the process ensuring our implementation fits our eventual goal of a faithful miliary simulation. For this advanced scenario, each agent is provided with three bullets, and at each timepoint they may either choose to move, shoot, or to not take any action. Other than capturing the enemy base, a team can win by eliminating all enemy agents.

6.3.6 Learning policies with RL. Since our approach is agnostic to any specific RL algorithm, for this work we chose to use the widely popular and versatile Deep Q learning (DQN) algorithm [48] for all of our experiments. Based on a specific application or domain, a suitable algorithm can be seamlessly used in place of DQN. In our implementation, we combine a shallow Q-Net architecture with techniques discussed in [48] such as experience replay, stable learning, and hard updates for the target network. In our architecture, we use one hidden layer between the input and output layers, 64 state variables (one for each grid cell), and an action space of 5 (for the base scenario) or 9 (for the advanced scenario). The observation state space available to the agent is symbolic in nature, and its size varied between experimental setups as follows:

Table 7. Example rules in first order logic and descriptions in natural language.

Rule Identifier	Rule	English Description
m_Down_on	$moveDown(A) : [1, 1] \xleftarrow[\Delta t=0]{} agent(A) : [1, 1] \wedge moveDir(A, down) : [1, 1] \wedge atLoc(A, X) : [1, 1] \wedge downLoc(Y, X) : [1, 1] \wedge blocked(Y) : [0, 0]$	If A is an agent (annotated $[1, 1]$) at location X , chooses to move in downward direction to Y (which is not blocked), then $moveDown(A)$'s label is updated to $[1, 1]$.
s_Left_on	$shootLeftB(A) : [1, 1] \xleftarrow[\Delta t=0]{} agent(A) : [1, 1] \wedge team(A, blue) : [1, 1] \wedge health(A) : [0.1, 1] \wedge ammo(A) : [0.1, 1] \wedge shootLeft(A) : [1, 1]$	If A is an agent on the blue team and chooses to shoot left, then $shootLeftB(A)$'s label is updated to $[1, 1]$ iff A has non-zero health and remaining ammo.

- (1) *Four* for single agent in the base scenario: two each for the current positions of the agent and the opponent.
- (2) *Seven* for single agent in the advanced scenario: one for the number of opponent bullets in the environment, two for the nearest bullet position, and two each for the current positions of the agent and the opponent.

For multi-agent setups, the observation space is multiplied by the number of agents in each team. For the special non-Markovian setup described later, the observation space is doubled as observations from previous the timestep are considered. For experiments in multi-agent environments, we learn non cooperative single agent policies using multi-agent sampling. We use the widely adopted Smooth L1 loss function, instead of gradient clipping as described in the seminal DQN work.

We use the following reward function (rewards related to shooting actions are only applicable to the advanced scenario):

- (1) Terminal state rewards: +250 for a win, -250 for a loss, +400 for shooting an opponent, -200 for getting shot.
- (2) Non-terminal state rewards: -2 for a valid action, -200 for an unsafe or illegal action, -10 for an invalid action (such as trying to shoot after exhausting ammunition).

We define the behavior of the opponent using a stochastic base policy, which at each timestep tries to move closer to the enemy base by reducing the manhattan distance with a probability of 0.7, or chooses a random action from the action space with a probability of 0.3. In the advanced scenario, shooting is prioritized over movement until ammo is exhausted. All RL policies described in this paper were learned on an NVIDIA A100 GPU with 80GB memory, and 40 cores of AMD EPYC 7413 with 378GB memory.

6.3.7 Shielding in RL. As discussed in Section 1, we incorporate logic shielding within the reward function, as well as the simulation environment itself. In the reward function, the agent is heavily penalized for taking an unsafe action, such as trying to move through the mountains or choosing an action that takes it out of bounds. While this approach encourages the agent to learn policies that avoid unsafe actions, it provides no guarantees. Adding shielding in the simulator itself ensures that even if the agent was to choose an unsafe action, our rule-based environment dynamics can detect and stop the execution of such actions in runtime. Furthermore, we can leverage these dynamics to prevent illegal actions, such as shooting when ammo has already been exhausted.

6.3.8 Exploring Limitations of the Markov Assumption. The Markov assumption in RL is the assumption that the next state of an agent only depends on its current state and action, and not on the history of states and actions that led to

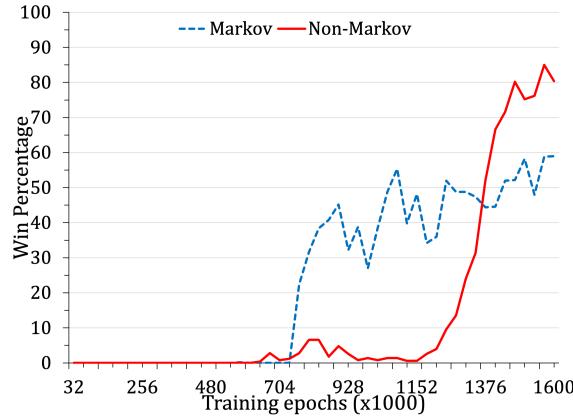


Fig. 14. Win percentage for policies learned with Markovian and non-Markovian dynamics.

the current state. As this simplifies the problem and enables the use of techniques like Markov Decision Processes (MDPs) and the Bellman equation, many well-established simulators make this assumption. However, many real-world environments are not truly Markovian, since in some cases the current state may not contain all the relevant information for decision-making. This is especially important for simulators replicating realistic military combat environments where various key factors like logistical support, conflict history, long-term intelligence data, and patterns in surveillance reports, which go into tactical decision making, are non-Markovian in nature.

PyReason does not make a Markov assumption, and we now showcase this capability by creating a simple experiment with non-Markovian dynamics. We consider a two-agents per team advanced scenario as described earlier. We introduce a modification to one agent within each team, constraining its ability to execute actions to once every two timesteps, with the added stipulation that each of its movement actions require two timesteps to complete. We learn to play the game in two different ways. In the initial approach, the player adheres to a Markov assumption, leveraging solely the current state's information. Conversely, in the second approach, the player gains access not only to the present state data but also to observations from the preceding time step. We compare the success of the two methods by evaluating learned policies over 500 games after every 32,000 training epochs.

Evolution of the performance of policies learned with and without the Markov assumption is shown in Figure 14. Both agents underwent training for a duration of up to 1.6 million epochs, with policy evaluations conducted at intervals of 32,000 epochs. Each policy was used to play the advanced scenario 500 times in order to obtain a win percentage. Evaluations were carried out on 48 cores of AMD EPYC 7413 with 378GB memory. Markovian policies obtained a peak performance of 59%, significantly lower than the 85% achieved by the non-Markovian policies. However, we observe that policies learned in the Markovian assumption setting attained decent performance with noticeably less training, which is unsurprising given the doubling of the observation space in the non-Markovian case. When examining the most effective policy within each category, the removal of the Markov assumption resulted in an increase in the average number of actions per agent required to secure a single victory, rising from 15.51 to 18.01. This observation suggests the acquisition of a policy characterized by greater complexity, yet one that exhibits enhanced reliability. Despite the relative simplicity of our experiment, a noteworthy performance enhancement was observed. This underscores the essentiality and significance of accommodating non-Markovian dynamics within simulation environments.

Win percentage over 500 trials, for policies learned with non-Markovian dynamics is shown in Figure 14. Each team is made up of one fast-moving and one slow-moving agent. The action space is extended to include two timesteps.

6.3.9 Explainability. One the major drawbacks of deep learning-based systems is the difficulty associated with understanding the output in terms of how it was computed. On the other hand, logic programs inherently support this kind of understanding. PyReason works with graphs using first order logical rules and produces an explainable trace detailing rules fired at different timesteps, constants used for grounding, and interpretation changes—an example is shown in Table 7. The explainable trace is a direct result of the structures leveraged in computational logic; this makes our approach explainable, allowing the user to understand system behavior and debug errors.

Two examples of how we leveraged this to improve our reward function given in Section 6.3.6 are:

- (1) Initially, we had set the penalty for getting shot at 400. However, from rule traces we observed that the agent was learning to prioritize hiding behind impenetrable mountains and take a safety first approach, instead of trying to win the game. Halving the penalty to 200 produced a more balanced policy.
- (2) The penalty for trying to shoot after exhausting ammunition was set to a lower value of 10 after observing that higher values led to the agent avoiding shooting altogether.

An excerpt of a rule trace is shown in Table 8; it begins at timestep 16 of one of our experiments. Initial conditions are as depicted in Figure 12. “R” and “B” respectively show the location of the red and blue agents at the beginning of this example. As the red agent moves downward from its starting location (from “24” to “0” through “16” and “8”), the blue agent decides to shoot to the left so as to intercept red (at “0”). However, red has seemingly learned to predict the bullet path and evade it. So it backtracks (to “16”). Rule **m_Down_on** presented in Table 7 is fired at timestep 16 (as well as 17 and 18), and is pictorially shown with a red arrow in Figure 12, and in bold in Table 8.

7 Conclusions and Future Work

This work introduces LAT logic, a logic programming framework that integrates temporal extensions with a lower lattice annotation structure to model non-Markovian temporal relationships while ensuring tractable and scalable exact reasoning. By departing from the standard Markov assumption, LAT logic enables reasoning about dependencies spanning multiple past time steps, capturing complex dynamic behaviors that traditional approaches like Markov Decision Processes cannot represent. Through rigorous theoretical analysis, we prove the correctness, convergence, and inconsistency detection capability of a fixpoint operator for this logic, and demonstrate how the use of a lower lattice facilitates Skolemization that significantly reduces the amount of grounding required. Our implementation, called PyReason, leverages these properties to efficiently perform reasoning over large-scale, sparse domains common in real-world applications such as multi-agent geospatial simulations, knowledge graph completion, and reinforcement learning. Empirically, we showed that LAT logic achieves multiple orders of magnitude improvements in grounding size, computational speed, and memory efficiency, making previously intractable reasoning tasks feasible. Moreover, by integrating non-Markovian dynamics into simulation environments, we show notable gains in reinforcement learning performance, highlighting the practical importance of tractable non-Markovian reasoning. This work thus bridges a critical gap by providing both a theoretically sound and practically scalable approach for modeling and reasoning about non-Markovian temporal dynamics in intelligent systems.

Looking ahead, several promising directions emerge for extending this framework. Incorporating probabilistic reasoning stands out as a natural next step, as APT logic [60], which combined temporal logic with probabilistic semantics, suffers from intractability. Leveraging recent advances in tractable probabilistic circuit learning, as developed by Choi et al.

Table 8. An extract of a rule trace produced by the PyReason software.

t	Constant Symbols	Predicate	Old Annotation	New Annotation	Rule fired
0	26	blocked	[0.0,1.0]	[1.0,1.0]	-
0	27	blocked	[0.0,1.0]	[1.0,1.0]	-
16	red-agent-1	moveDown	[0.0,0.0]	[1.0,1.0]	m_Down_on
17	red-agent-1	moveDown	[1.0,1.0]	[0.0,0.0]	m_Down_off
17	(red-agent-1,16)	atLoc	[0.0,1.0]	[1.0,1.0]	m_Set_location
17	(red-agent-1,24)	atLoc	[1.0,1.0]	[0.0,0.0]	m_Rem_location
17	red-agent-1	moveDown	[0.0,0.0]	[1.0,1.0]	m_Down_on
18	red-agent-1	moveDown	[1.0,1.0]	[0.0,0.0]	m_Down_off
18	(red-agent-1,8)	atLoc	[0.0,1.0]	[1.0,1.0]	m_Set_location
18	(red-agent-1,16)	atLoc	[1.0,1.0]	[0.0,0.0]	m_Rem_location
18	blue-agent-1	shootLeftB	[0.0,1.0]	[1.0,1.0]	s_Left_on
18	(blue-bullet-1,3)	atLoc	[0.0,1.0]	[1.0,1.0]	s_Set_location
18	(blue-bullet-1,left)	direction	[0.0,1.0]	[1.0,1.0]	s_Set_dir
18	red-agent-1	moveDown	[0.0,0.0]	[1.0,1.0]	m_Down_on
19	red-agent-1	moveDown	[1.0,1.0]	[0.0,0.0]	m_Down_off
19	(red-agent-1,0)	atLoc	[0.0,1.0]	[1.0,1.0]	m_Set_location
19	(red-agent-1,8)	atLoc	[1.0,1.0]	[0.0,0.0]	m_Rem_location
19	blue-agent-1	shootLeftB	[1.0,1.0]	[0.0,0.0]	s_Left_off
19	(blue-bullet-1,3)	atLoc	[1.0,1.0]	[0.0,0.0]	s_Rem_location
19	(blue-bullet-1,2)	atLoc	[0.0,1.0]	[1.0,1.0]	s_Set_location
19	red-agent-1	moveUp	[0.0,0.0]	[1.0,1.0]	m_Up_on
20	red-agent-1	moveUp	[1.0,1.0]	[0.0,0.0]	m_Up_off
20	(red-agent-1,8)	atLoc	[0.0,0.0]	[1.0,1.0]	m_Set_location
20	(red-agent-1,0)	atLoc	[1.0,1.0]	[0.0,0.0]	m_Rem_location
20	(blue-bullet-1,2)	atLoc	[1.0,1.0]	[0.0,0.0]	s_Rem_location
20	(blue-bullet-1,1)	atLoc	[0.0,1.0]	[1.0,1.0]	s_Set_location
20	red-agent-1	moveUp	[0.0,0.0]	[1.0,1.0]	m_Up_on
21	red-agent-1	moveUp	[1.0,1.0]	[0.0,0.0]	m_Up_off
21	(red-agent-1,16)	atLoc	[0.0,0.0]	[1.0,1.0]	m_Set_location
21	(red-agent-1,8)	atLoc	[1.0,1.0]	[0.0,0.0]	m_Rem_location
21	(blue-bullet-1,1)	atLoc	[1.0,1.0]	[0.0,0.0]	s_Rem_location
21	(blue-bullet-1,0)	atLoc	[0.0,1.0]	[1.0,1.0]	s_Set_location

al. [15], could enable learning probability distributions within LAT logic’s efficient and tractable semantics, allowing a rich yet computationally feasible representation of uncertainty beyond deterministic intervals. Another potential avenue is constructing logic programs using large language models, as introduced in Logic LM [52], which demonstrated the use of LLMs with symbolic solvers for reasoning tasks—extensions to temporal logic remain unexamined in this front. Finally, applying Inductive Logic Programming approaches [28] to automatically learn temporal and non-Markovian rules within LAT logic offers a compelling route to scale and adapt the framework to data-driven scenarios where expert knowledge is limited or unavailable, providing a pathway to fully automated, explainable temporal reasoning systems. PyReason has already been successfully applied in several domains, including reasoning about medical triage optimization [53], integrating machine learning models with temporal logic for process automation [2], as well as

abductive reasoning in vision [41] and geospatial applications [6]. A promising direction for future work is to explore abductive queries in a more general way, further enhancing the framework’s reasoning capabilities and applicability across diverse problems.

Acknowledgments

Some of the authors were funded by Scientific Systems Company, Inc. (SSCI).

References

- [1] Dyuman Aditya, Kaustuv Mukherji, Srikanth Balasubramanian, Abhiraj Chaudhary, and Paulo Shakarian. 2023. PyReason: Software for Open World Temporal Logic. In *AAAI Spring Symposium: MAKE*.
- [2] Dyuman Aditya, Colton Payne, Mario Leiva, and Paulo Shakarian. 2025. Machine Learning Model Integration with Open World Temporal Logic for Process Automation. *arXiv preprint arXiv:2506.17776* (2025).
- [3] Mridul Agarwal and Vaneet Aggarwal. 2023. Reinforcement learning for joint optimization of multiple rewards. *Journal of Machine Learning Research* 24, 49 (2023), 1–41.
- [4] Kiyoshi Akama and Ekawit Nantajeewarawat. 2011. Meaning-preserving skolemization. In *International Conference on Knowledge Engineering and Ontology Development*, Vol. 2. SCITEPRESS, 322–327.
- [5] Claudi Alsina, Enric Trillas, and Llorenç Valverde. 1983. On some logical connectives for fuzzy sets theory. *J. Math. Anal. Appl.* 93, 1 (1983), 15–26.
- [6] D Bavikadi, D Aditya, D Parkar, P Shakarian, G Mueller, C Parvis, and GI Simari. 2025. Geospatial Trajectory Generation via Efficient Abduction: Deployment for Independent Testing. *ELECTRONIC PROCEEDINGS IN THEORETICAL COMPUTER SCIENCE* 416, 416 (2025), 274–287.
- [7] Divyagna Bavikadi, Nathaniel Lee, Paulo Shakarian, and Chad Parvis. [n. d.] Sea-cret Agents: Maritime Abduction for Region Generation to Expose Dark Vessel Trajectories. ([n. d.]).
- [8] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. 2013. Translating Embeddings for Modeling Multi-relational Data. In *Advances in Neural Information Processing Systems*, C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger (Eds.), Vol. 26. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2013/file/1cecc7a77928ca8133fa24680a88d2f9-Paper.pdf
- [9] Laura Bozzelli and David Pearce. 2015. On the Complexity of Temporal Equilibrium Logic. In *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science*. 645–656. doi:[10.1109/LICS.2015.65](https://doi.org/10.1109/LICS.2015.65)
- [10] Andreas Bueff and Vaishak Belle. 2023. Deep inductive logic programming meets reinforcement learning. *arXiv preprint arXiv:2308.16210* (2023).
- [11] Andreas Burgdorf, Alexander Paulus, André Pomp, and Tobias Meisen. 2022. DocSemMap: Leveraging Textual Data Documentations for Mapping Structured Data Sets into Knowledge Graphs. In *2022 IEEE 16th International Conference on Semantic Computing (ICSC)*. 209–216. doi:[10.1109/ICSC52841.2022.00042](https://doi.org/10.1109/ICSC52841.2022.00042)
- [12] Pedro Cabalar and Gilberto Pérez Vega. 2007. Temporal Equilibrium Logic: A First Approach. In *Computer Aided Systems Theory – EUROCAST 2007*, Roberto Moreno Díaz, Franz Pichler, and Alexis Quesada Arencibia (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 241–248.
- [13] Stefano Ceri, Georg Gottlob, Letizia Tanca, et al. 1989. What you always wanted to know about Datalog (and never dared to ask). *IEEE transactions on knowledge and data engineering* 1, 1 (1989), 146–166.
- [14] Siddharth Chandak, Pratik Shah, Vivek S Borkar, and Parth Dodhia. 2024. Reinforcement learning in non-Markovian environments. *Systems & Control Letters* 185 (2024), 105751.
- [15] Y Choi, Antonio Vergari, and Guy Van den Broeck. 2020. Probabilistic circuits: A unifying framework for tractable probabilistic models. *UCLA*. URL: <http://starai.cs.ucla.edu/papers/ProbCirc20.pdf> (2020), 6.
- [16] Edmund M Clarke, E Allen Emerson, and A Prasad Sistla. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8, 2 (1986), 244–263.
- [17] Rance Cleaveland, S. Purushothaman Iyer, and Murali Narasimha. 2005. Probabilistic temporal logics via the modal mu-calculus. *Theoretical Computer Science* 342, 2 (2005), 316–350. doi:[10.1016/j.tcs.2005.03.048](https://doi.org/10.1016/j.tcs.2005.03.048)
- [18] Peter D Clive, Jeffrey A Johnson, Michael J Moss, James M Zeh, Brian M Birkmire, and Douglas D Hodson. 2015. Advanced framework for simulation, integration and modeling (AFSIM)(Case Number: 88ABW-2015-2258). In *Proceedings of the international conference on scientific computing (CSC)*. The Steering Committee of The World Congress in Computer Science, Computer ..., 73.
- [19] Alain Colmerauer. 1982. Prolog and infinite trees. *Logic Programming* 16, 231–251 (1982), 2.
- [20] Maxwell John Cresswell and George Edward Hughes. 2012. *A new introduction to modal logic*. Routledge.
- [21] Andrew Cropper, Sebastian Dumančić, and Stephen H Muggleton. 2020. Turning 30: New ideas in inductive logic programming. *arXiv preprint arXiv:2002.11002* (2020).
- [22] Wang-Zhou Dai, Qiuling Xu, Yang Yu, and Zhi-Hua Zhou. 2019. Bridging machine learning and logical reasoning by abductive learning. *Advances in Neural Information Processing Systems* 32 (2019).
- [23] Alex Dekhtyar, Michael I Dekhtyar, and VS Subrahmanian. 1999. Temporal Probabilistic Logic Programs.. In *ICLP*, Vol. 99. 109–123.

- [24] Alex Dekhtyar, Michael I. Dekhtyar, and V. S. Subrahmanian. 1999. Temporal Probabilistic Logic Programs. In *International Conference on Logic Programming*. 109–123.
- [25] Martin Diller, Adam Wyner, and Hannes Strass. 2019. Making Sense of Conflicting (Defeasible) Rules in the Controlled Natural Language ACE: Design of a System with Support for Existential Quantification Using Skolemization. In *Proceedings of the 13th International Conference on Computational Semantics - Short Papers*, Simon Dobnik, Stergios Chatzikyriakidis, and Vera Demberg (Eds.). Association for Computational Linguistics, Gothenburg, Sweden, 32–37. doi:[10.18653/v1/W19-0505](https://doi.org/10.18653/v1/W19-0505)
- [26] Dragan Doder and Zoran Ognjanović. 2024. Probabilistic temporal logic with countably additive semantics. *Annals of Pure and Applied Logic* 175, 9 (2024), 103389.
- [27] Esra Erdem, Michael Gelfond, and Nicola Leone. 2016. Applications of answer set programming. *Ai Magazine* 37, 3 (2016), 53–68.
- [28] Richard Evans and Edward Grefenstette. 2018. Learning Explanatory Rules from Noisy Data. *J. Artif. Int. Res.* 61, 1 (jan 2018), 1–64.
- [29] Hayden Freedman, Neda Abolhassani, Jacob Metzger, and Sanjoy Paul. 2023. Ontology Modeling for Probabilistic Knowledge Graphs. In *2023 IEEE 17th International Conference on Semantic Computing (ICSC)*. 252–259. doi:[10.1109/ICSC56153.2023.00049](https://doi.org/10.1109/ICSC56153.2023.00049)
- [30] Maor Gaon and Ronen Brafman. 2020. Reinforcement learning with non-markovian rewards. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 3980–3987.
- [31] Matt Ginsberg. 2012. *Essentials of Artificial Intelligence*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- [32] M Gelfond GL881 and V Lifschitz. 1988. The stable model semantics for logic programming. In *Proc. 5'h International Conference and Symposium on Logic Programming*. 1070–1080.
- [33] Gaurav Gupta, Chenzhong Yin, Jyotirmoy V Deshmukh, and Paul Bogdan. 2021. Non-markovian reinforcement learning using fractional dynamics. In *2021 60th IEEE Conference on Decision and Control (CDC)*. IEEE, 1542–1547.
- [34] Hans Hansson and Bengt Jonsson. 1994. A logic for reasoning about time and reliability. *Formal aspects of computing* 6, 5 (1994), 512–535.
- [35] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.
- [36] Patrick Hohenegger and Thomas Lukasiewicz. 2020. Ontology reasoning with deep neural networks. In *Journal of Artificial Intelligence Research*, Vol. 68. 503–540.
- [37] Ulrich Höhle. 1978. Probabilistic uniformization of fuzzy topologies. *Fuzzy Sets and Systems* (1978).
- [38] Michael Kifer and V.S. Subrahmanian. 1992. Theory of Generalized Annotated Logic Programming and its Applications. *J. Log. Program.* 12, 3&4 (1992), 335–367.
- [39] Robert A. Kowalski. 1988. The early years of logic programming. *Commun. ACM* 31, 1 (Jan. 1988), 38–43. doi:[10.1145/35043.35046](https://doi.org/10.1145/35043.35046)
- [40] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. 75–86. doi:[10.1109/CGO.2004.1281665](https://doi.org/10.1109/CGO.2004.1281665)
- [41] Mario Leiva, Noel Ngu, Joshua Shay Kricheli, Aditya Taparia, Ransalu Senanayake, Paulo Shakarian, Nathaniel Bastian, John Corcoran, and Gerardo Simari. 2025. Consistency-based Abductive Reasoning over Perceptual Errors of Multiple Pre-trained Models in Novel Environments. *arXiv preprint arXiv:2505.19361* (2025).
- [42] D Loveland. 1978. Automated Theorem Proving: A Logical Basis North Holland. New York (1978), N75.
- [43] Victor W Marek and Mirosław Truszczyński. [n. d.]. Stable models and an alternative logic programming paradigm. In *The logic programming paradigm: A 25-year perspective*. Springer, 375–398.
- [44] Joao Marques-Silva. 2024. Logic-based explainability: past, present and future. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 181–204.
- [45] Karsten Martiny and Ralf Möller. 2016. PDT logic: a probabilistic doxastic temporal logic for reasoning about beliefs in multi-agent systems. *Journal of Artificial Intelligence Research* 57 (2016), 39–112.
- [46] Bridget T McInnes, Ted Pedersen, and Serguei VS Pakhomov. 2009. UMLS-Interface and UMLS-Similarity: open source software for measuring paths and semantic similarity. In *AMIA annual symposium proceedings*, Vol. 2009. American Medical Informatics Association, 431.
- [47] Christian Meilicke, Melisachew Wudage Chekol, Patrick Betz, Manuel Fink, and Heiner Stuckeschmidt. 2024. Anytime bottom-up rule learning for large-scale knowledge graph completion. *The VLDB Journal* 33, 1 (2024), 131–161.
- [48] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *nature* 518, 7540 (2015), 529–533.
- [49] Charles G Morgan. 1974. Symbolic Logic and Mechanical Theorem Proving (Chin-Liang Chang and Richard Char-Tung Lee). *SIAM Rev.* 16, 3 (1974), 403–407.
- [50] Kaustuv Mukherji, Devendra Parkar, Lahari Pokala, Dyuman Aditya, Paulo Shakarian, and Clark Dorman. 2024. Scalable Semantic Non-Markovian Simulation Proxy for Reinforcement Learning. In *2024 IEEE 18th International Conference on Semantic Computing (ICSC)*. IEEE, 183–190.
- [51] Ilkka Niemelä. 1999. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. *Ann. Math. Artif. Intell.* 25 (11 1999), 241–273. doi:[10.1023/A:1018930122475](https://doi.org/10.1023/A:1018930122475)
- [52] Liangming Pan, Alon Albala, Xinyi Wang, and William Yang Wang. [n. d.]. Logic-LM: Empowering Large Language Models with Symbolic Solvers for Faithful Logical Reasoning. In *The 2023 Conference on Empirical Methods in Natural Language Processing*.
- [53] Jaikrishna Manojkumar Patil, Adam Chapman, Richard Knuszka, John Chapman, and Paulo Shakarian. 2025. Reasoning about Medical Triage Optimization with Logic Programming. *arXiv preprint arXiv:2507.10781* (2025).

- [54] David Pearce. 2006. Equilibrium logic. *Annals of Mathematics and Artificial Intelligence* 47, 1–2 (June 2006), 3–41. doi:10.1007/s10472-006-9028-z
- [55] Amir Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 46–57. doi:10.1109/SFCS.1977.32
- [56] Ryan Riegel, Alexander Gray, Francois Luus, Naweed Khan, Ndivhuwo Makondo, Ismail Yunus Akhalwaya, Haifeng Qian, Ronald Fagin, Francisco Barahona, Udit Sharma, Shajith Iqbal, Hima Karanam, Sumit Neelam, Ankita Likhyani, and Santosh Srivastava. 2020. Logical Neural Networks.
- [57] Jürgen Schmidhuber. 1990. Reinforcement learning in Markovian and non-Markovian environments. *Advances in neural information processing systems* 3 (1990).
- [58] Lenhart Schubert. 1999. Dynamic Skolemization. In *Computing Meaning: Volume 1*. Springer, 219–253.
- [59] Prithviraj Sen, Breno W. S. R. de Carvalho, Ryan Riegel, and Alexander Gray. 2022. Neuro-Symbolic Inductive Logic Programming with Logical Neural Networks. *AAAI conference on Artificial Intelligence* 8 (2022).
- [60] Paulo Shakarian, Austin Parker, Gerardo I. Simari, and Venkatramana V. S. Subrahmanian. 2011. Annotated probabilistic temporal logic. *ACM Trans. Comput. Logic* 12, 2, Article 14 (jan 2011), 44 pages. doi:10.1145/1877714.1877720
- [61] Paulo Shakarian and Gerardo I Simari. 2022. Extensions to Generalized Annotated Logic and an Equivalent Neural Architecture. In *2022 Fourth International Conference on Transdisciplinary AI (TransAI)*. IEEE, 63–70.
- [62] Paulo Shakarian, Gerardo I. Simari, and Robert Schroeder. 2013. MANCaLog: a logic for multi-attribute network cascades. In *International conference on Autonomous Agents and Multi-Agent Systems, AAMAS*. 1175–1176.
- [63] Paulo Shakarian, Gerardo I Simari, and VS Subrahmanian. 2012. Annotated probabilistic temporal logic: Approximate fixpoint implementation. *ACM Transactions on Computational Logic (TOCL)* 13, 2 (2012), 1–33.
- [64] Hikaru Shindo, Masaaki Nishino, and Akihiro Yamamoto. 2021. Differentiable Inductive Logic Programming for Structured Examples. In *AAAI Conference on Artificial Intelligence*. 5034–5041.
- [65] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. 2007. Yago: a core of semantic knowledge. In *Proceedings of the 16th International Conference on World Wide Web* (Banff, Alberta, Canada) (*WWW '07*). Association for Computing Machinery, New York, NY, USA, 697–706. doi:10.1145/1242572.1242667
- [66] Atiya Usmani, Saeed Hamood Alsamhi, John Breslin, and Edward Curry. 2023. A Novel Framework for Constructing Multimodal Knowledge Graph from MuSe-CaR Video Reviews. In *2023 IEEE 17th International Conference on Semantic Computing (ICSC)*. 323–328. doi:10.1109/ICSC56153.2023.00066
- [67] Alexandros Vassiliades, Spyridon Symeonidis, Sotiris Diplaris, Georgios Tzanetis, Stefanos Vrochidis, Nick Bassiliades, and Ioannis Kompatsiaris. 2023. XR4DRAMA Knowledge Graph: A Knowledge Graph for Disaster Management. In *2023 IEEE 17th International Conference on Semantic Computing (ICSC)*. 262–265. doi:10.1109/ICSC56153.2023.00051
- [68] A Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* (2017).
- [69] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. 2017. Starcraft ii: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782* (2017).
- [70] Peter Vojtáš. 2001. Fuzzy logic programming. *Fuzzy sets and systems* 124, 3 (2001), 361–370.

A Complete Proof for Theorem 4.4

PROOF. Let $P_{\Pi} \subseteq P$ be the set of predicates containing only predicates present in the head of at least one rule in Π_{Rules} .

$$\begin{aligned}
 |g_i| &= |\bigcup_{p \in P} g_i(p)| \\
 &= \sum_{p \in P} |g_i(p)| \\
 &= \sum_{p \in P_{\Pi}} |g_i(p)| + \sum_{p \notin P_{\Pi}} |g_i(p)| \\
 &= \sum_{p \in P_{\Pi}} |g_i(p)| + \sum_{p \notin P_{\Pi}} |g_0(p)|
 \end{aligned} \tag{12}$$

Let, $\Gamma_r(g)$ denote the set of ground atoms produced when a single fixpoint operator is applied to a single rule r with the set of ground atoms g .

$$\begin{aligned}
 |g_i(p)| &= |g_{i-1}(p) \cup \bigcup_{r \in \Pi_{rules} \wedge pred(head(r))=p} \Gamma_r(g_{i-1})| \\
 &= |g_{i-1}(p)| + newF_{p,i} \times \left| \bigcup_{r \in \Pi_{rules} \wedge pred(head(r))=p} \Gamma_r(g_{i-1}) \right| \\
 &= |g_{i-1}(p)| + newF_{p,i} \times uniqueF_{p,i} \times \sum_{r \in \Pi_{rules} \wedge pred(head(r))=p} |\Gamma_r(g_{i-1})|
 \end{aligned} \tag{13}$$

Here, $newF_{p,i} \in [0, 1]$ denotes the fraction of ground atoms produced, with predicate p and at the i^{th} Γ application, which did not exist after the $(i-1)^{th}$ application. Similarly, $uniqueF_{p,i} \in [0, 1]$ is the fraction of ground atoms produced across rules, with predicate p in the head, which are unique.

$$\begin{aligned}
 |\Gamma_r(g_{i-1})| &\leq \prod_j |g_{i-1}(pred(body(r), j))| \\
 |\Gamma_r(g_{i-1})| &= validF_{r,i} \times \prod_j |g_{i-1}(pred(body(r), j))|
 \end{aligned} \tag{14}$$

Here, $validF_{r,i} \in [0, 1]$ denotes the fraction of valid groundings that leads to firing of non-ground rule r , within the cross-product of possible groundings for each body clause.

From Eqs. (13) and (14) we get:

$$|g_i(p)| = |g_{i-1}(p)| + newF_{p,i} \times uniqueF_{p,i} \sum_{\substack{r \in \Pi_{rules} \\ pred(head(r))=p}} validF_{r,i} \times \prod_j |g_{i-1}(pred(body(r), j))| \tag{15}$$

$$|g_i(p)| - |g_{i-1}(p)| = newF_{p,i} \times uniqueF_{p,i} \sum_{\substack{r \in \Pi_{rules} \\ pred(head(r))=p}} validF_{r,i} \times \prod_j |g_{i-1}(pred(body(r), j))| \tag{16}$$

$$\Delta|g_i(p)| = newF_{p,i} \times uniqueF_{p,i} \sum_{\substack{r \in \Pi_{rules} \\ pred(head(r))=p}} validF_{r,i} \times \prod_j |g_{i-1}(pred(body(r), j))|$$

Considering the maximum value ($= 1$) for all three fractions:

$$\Delta|g_i(p)| \leq \sum_{\substack{r \in \Pi_{rules} \\ pred(head(r))=p}} \prod_j |g_{i-1}(pred(body(r), j))|$$

Substituting Equation (15) into Equation (12) we obtain:

$$|g_i| = \sum_{p \in P_\Pi} \left[|g_{i-1}(p)| + newF_{p,i} \times uniqueF_{p,i} \sum_{\substack{r \in \Pi_{rules} \\ pred(head(r))=p}} validF_{r,i} \prod_j |g_{i-1}(pred(body(r), j))| \right] + \sum_{p \notin P_\Pi} |g_0(p)|$$

Table 9. PyReason configuration settings for geospatial experiments.

Setting	Value	Description
<i>verbose</i>	True	Print all info to screen during reasoning.
<i>atom_trace</i>	False	Groundings untracked, reducing overhead for large graphs.
<i>persistent</i>	False	Interpretations are not reset to bottom of the lattice after every timestep.
<i>static_graph_facts</i>	False	Interpretations in the input graph are allowed to change during reasoning.
<i>parallel_computing</i>	True	Use parallel processing.
<i>ad_hoc_grounding</i>	True	Use skolemization.
<i>resolution_levels</i>	2,3,4,5,6,7,8,9	Grid Size = $(2^{resolution_levels})^2$.

$$|g_i| = \sum_{p \in P_\Pi} |g_{i-1}(p)| + \sum_{p \notin P_\Pi} |g_0(p)| \\ + \sum_{p \in P_\Pi} \left[newF_{p,i} \times uniqueF_{p,i} \sum_{\substack{r \in \Pi_{rules} \\ pred(head(r))=p}} validF_{r,i} \times \prod_j |g_{i-1}(pred(body(r), j))| \right] \quad (17)$$

$$|g_i| = |g_{i-1}| + \sum_{p \in P_\Pi} newF_{p,i} \times uniqueF_{p,i} \sum_{\substack{r \in \Pi_{rules} \\ pred(head(r))=p}} validF_{r,i} \times \prod_j |g_{i-1}(pred(body(r), j))|$$

$$\Delta|g_i| = \sum_{p \in P_\Pi} newF_{p,i} \times uniqueF_{p,i} \sum_{\substack{r \in \Pi_{rules} \\ pred(head(r))=p}} validF_{r,i} \prod_j |g_{i-1}(pred(body(r), j))| \quad (18)$$

Considering the maximum value for all three fractions:

$$\Delta|g_i| \leq \sum_{p \in P_\Pi} \sum_{\substack{r \in \Pi_{rules} \\ pred(head(r))=p}} \prod_j |g_{i-1}(pred(body(r), j))|$$

which further simplifies to:

$$\Delta|g_i| \leq \sum_{r \in \Pi_{rules}} \prod_j |g_{i-1}(pred(body(r), j))| \quad (19)$$

□

B Reproducibility guide

All experiments carried out to obtain the results shown in Section 6 use the PyReason framework with specific configurations. Geospatial experiments use the PyReason configuration settings shown in Table 9. The parameter *ad_hoc_grounding* acts as the key change between the Skolemization-enabled approach and the traditional full grounding approach. For knowledge graph completion experiments, we use the same PyReason configuration settings as presented in Table 9 with the exception that we only use *ad_hoc_grounding* = *False* since this refers to node-level Skolemization which we do not need for knowledge graph completion experiments. Thus, the *resolution_levels* parameter is not applicable for Knowledge graph completion tasks. Additionally, knowledge graph completion experiments used AnyBURL for learning rules with rule snapshot at twenty minutes for all datasets. The following are the commands to execute Python scripts for geospatial and knowledge graph completion experiments.

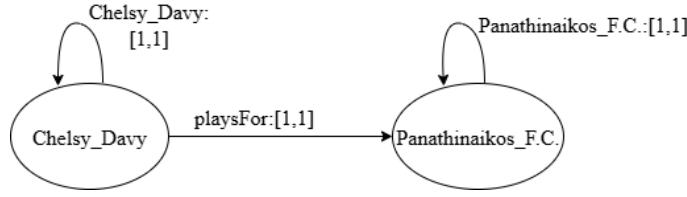


Fig. 15. Graph representation of example triple to work with PyReason.

B.1 Skolemization approach

```
python3 play_random_game.py --resolution 5 --field_soldiers_per_team 10
--border_soldiers_per_team 10 --actions_per_soldier 100 --ad_hoc
```

B.2 Full grounding approach

```
python3 play_random_game.py --resolution 5 --field_soldiers_per_team 10
--border_soldiers_per_team 10 --actions_per_soldier 100
```

B.3 Knowledge Graph completion

```
python3 anyBurl_multistep_multirule.py -rf yago_1200_99_100_ann
-s 1 -e 1000 -ts 10 -g anyBurl_graphs/YAGO3-10/knowledge_graph_train.graphml
```

C Example pipeline using PyReason

C.1 Knowledge Graph completion

Both the train and test set of any knowledge graph are set of triples. We show the example triple from the train set of the YAGO03-10 dataset that we need for our example as follows.

Chelsy_Davy playsFor Panathinaikos_F.C.

The next step is to convert the training triples into graphs that can be inputted into PyReason. An example of such a graph representation is shown in Figure 15.

We then convert the learned AnyBURL rules into PyReason rules. The sample AnyBURL rule used in our example is as follows:

```
0.934 isAffiliatedTo(X, Panathinaikos_F.C.) <= playsFor(X, Panathinaikos_F.C.)
```

The first number is the confidence value of the rule, which becomes the lower bound in the converted rule. Further, partially ground rules are made fully non-grounded to make grounding faster. The converted non-ground PyReason rule is then,

```
isAffiliatedTo(X, X_0) : [0.934, 1] <-1 playsFor(X, X_0) : [0.1, 1],
Panathinaikos_F.C.(X_0) : [1, 1]
```

After inference, an additional edge is added to the graph. The updated graph after inference is shown in Figure 16.

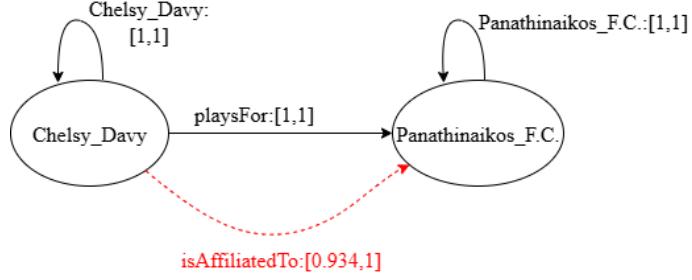


Fig. 16. Graph representation of example triple after Inference.

12	13	14	15
8	9	10	11
4	5	6	7
0	1	2	3

Fig. 17. Example grid for agent movement in Geospatial application. In this case, note that cell #5 has an obstacle, hence the agent is not allowed to move there.

C.2 Geospatial Application

For the geospatial skolemization experiment, we convert a grid map shown in Figure 17 into a graph structure shown in Figure 18 (a). Note that here we consider a grid map with the least grid size, i.e., 16, with the initial graph passed to PyReason in the case of Skolemization. For the non-Skolemization case, the complete graph with all grid points needs to be passed to PyReason. The following two non-ground rules are fired when the policy wants the agent to move in the right direction on the border of the grid:

```

atLoc(AGENT, NEWLOC) : [1,1] <-2 moveRight(AGENT) : [1,1], borderAgent(AGENT) : [1,1],
                           atLoc(AGENT, OLDLOC) : [1,1], right(OLDLOC, NEWLOC) : [1,1],
                           borderLoc(NEWLOC) : [1,1], blocked(NEWLOC) : [0,0]

atLoc(AGENT, OLDLOC) : [0,0] <-2 moveRight(AGENT) : [1,1], borderAgent(AGENT) : [1,1],
                           atLoc(AGENT, OLDLOC) : [1,1], right(OLDLOC, NEWLOC) : [1,1],
                           borderLoc(NEWLOC) : [1,1], blocked(NEWLOC) : [0,0]

```

Figure 18 (b) shows the resulting graph after two timesteps of reasoning.

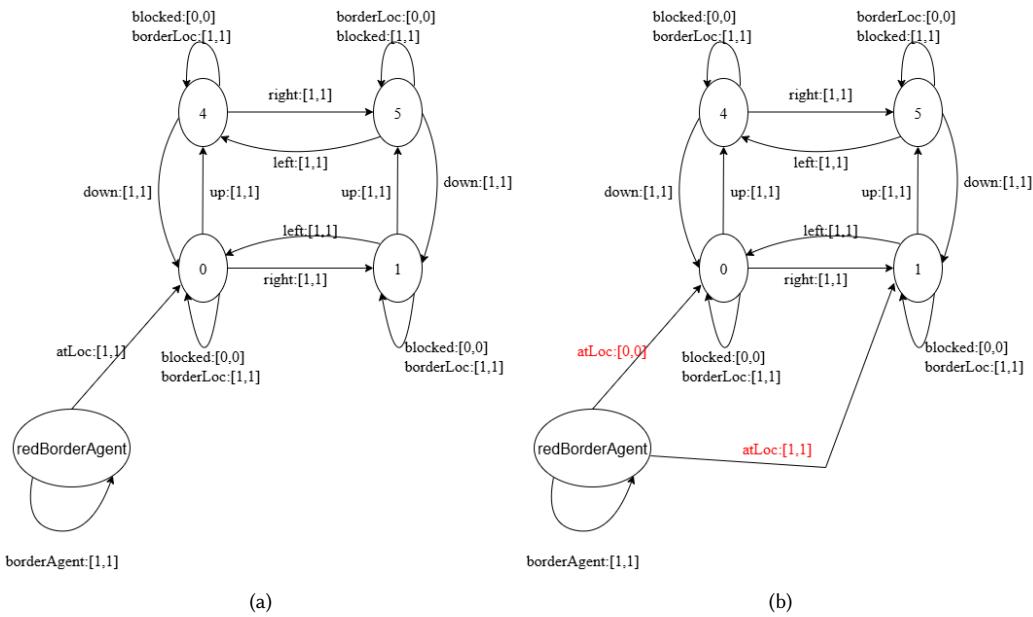


Fig. 18. (a) Initial PyReason graph representation of grid and agent location. (b) Updated PyReason graph after two timesteps of inference.