`Rules2Lab`: from Prolog Knowledge-Base, to Learning Agents, to Norm Engineering

Peter Fratrič*[1] [0000−0001−6378−5886], Nils Holzenberger[2] [0000−0002−0844−1391], and David Restrepo Amariles[1] [0000−0002−2841−2563]

[1] HEC Paris, France
[2] Télécom Paris, Institut Polytechnique de Paris, France

Abstract. Multi-agent environments are one of the main tools used in AI research. This paper proposes a methodology, called `Rules2Lab`, that maps an expert system into executable simulation laboratory, where interactions between agents and rules can be analyzed. States, actions, and constraints are defined using first-order logic implemented in Prolog, while temporal, computational and sub-symbolic operations of the lab environment are delegated to Python. This combination brings state-of-the-art machine learning and reinforcement learning libraries directly into the rich area of logic-based technologies for multi-agent systems. We demonstrate this approach through a case study addressing privacy vulnerabilities in a data marketplace, where regulations are defined as first-order predicates. The simulation involves agents attempting to access sensitive data, with privacy breach defined by similarity between inferred and private data. A reinforcement learning agent is trained to identify policies leading to privacy-breaching states. Subsequently, inductive logic programming is employed to derive norms that prevent privacy breaches, transforming ex-post enforcement into ex-ante compliance. The results showcase the feasibility of integrating Python for sub-symbolic processing with Prolog for logic-based modelling and inference, providing a flexible development paradigm for agent-based simulations.

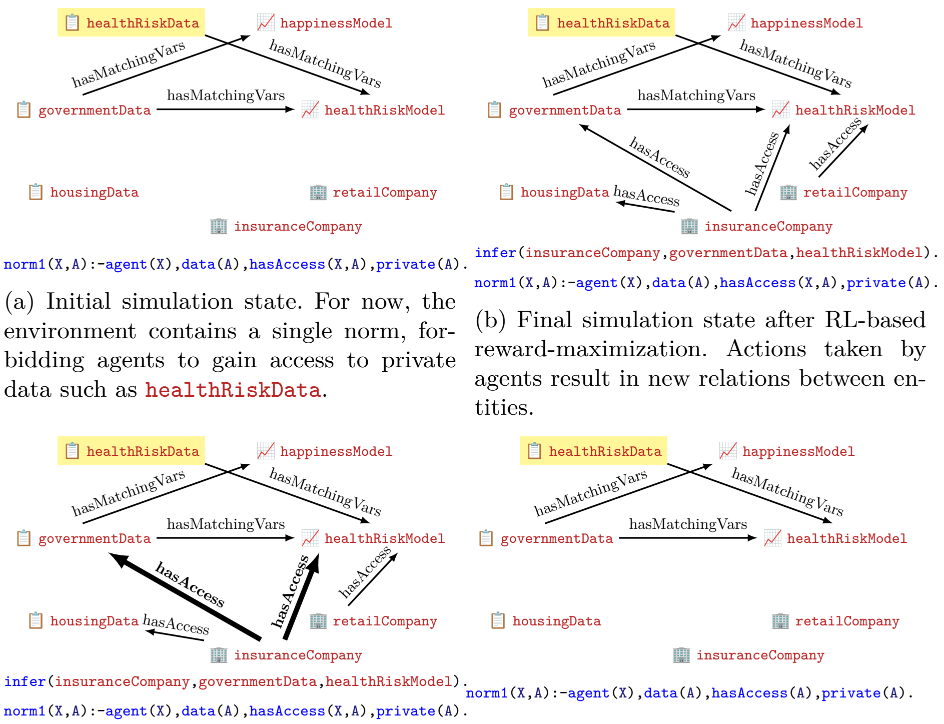
Keywords: multi-agent system · norms · reinforcement learning · expert systems · inductive logic programming

1 Introduction

Current predictive AI, whether based on expert systems or statistical learning, typically deals with single-step inference. Examples include predicting the occurrence of cancer from an X-ray, product recommendation from a user's attributes, and automatic detection of hate speech. In contrast, some predictions require many dependent computations: here, we are concerned with predicting how reward-based agents will react to rules constraining their behavior. As a running example, illustrated in Figure 1, we will consider how agents might find ways to access private data through a combination of publicly available data and

* Corresponding author: `fratric@hec.fr`

inference models. The aim is to identify weaknesses or loopholes in regulatory systems, before they can be exploited.



```
norm1(X,A):-agent(X),data(A),hasAccess(X,A),private(A).
```

(a) Initial simulation state. For now, the environment contains a single norm, forbidding agents to gain access to private data such as `healthRiskData`.

```
infer(insuranceCompany,governmentData,healthRiskModel).
norm1(X,A):-agent(X),data(A),hasAccess(X,A),private(A).
```

(b) Final simulation state after RL-based reward-maximization. Actions taken by agents result in new relations between entities.

```
infer(insuranceCompany,governmentData,healthRiskModel).
norm1(X,A):-agent(X),data(A),hasAccess(X,A),private(A).
```

(c) Rule induction identifies actions leading to privacy breach (thick lines).

```
norm1(X,A):-agent(X),data(A),hasAccess(A),private(A).
norm2(X,A,B):- agent(X),data(A),hasAccess(X,A),
    data(B),hasAccess(X,B),performedInference(X,A,B),
    hasPrivateAttributes(A),hasPrivateAttributes(B).
```

(d) New initial state for the next simulation, with additional norm identified by rule induction.

Fig. 1: One iteration of the simulation involving 2 agents 🏢, 3 instances of data 📋, and 2 instances of models 📈. In (b), the `insuranceCompany` gains access to data `governmentData` and model `healthRiskModel`, and uses it to infer the private `healthRiskData`. Once this unforeseen strategy of privacy breach has been identified in (c), a new norm is added to the next iteration of the simulation in (d).

Motivation Agent-based modelling is a research area where both symbolic and sub-symbolic technologies made considerable advancements. Sub-symbolic methods, such as deep reinforcement learning, reached high performance in several problem areas [24]. Yet, this performance increase typically came at a cost of transparency. Observability, interpretability, explainability, and accountability, are aspects inherent to logic-based technologies used for agent-based programming, and it can be argued that symbolic methods have the potential to be used for explaining behavior of agents lacking these attributes [11, 12, 31]. Whether we want to analyze behavior of agents through symbolic methods, or use statistics to explain behavior of symbolic multi-agent system (MAS), it is desirable to develop a common ground both for symbolic and sub-symbolic methods, that can be used to further advance research on multi-agent systems.

Contribution We address this issue by introducing a new approach to MAS development, called `Rules2Lab`, intended for studying agent behavior in a laboratory setting defined by an expert system.[3] Therefore, the main contribution of this paper lies in facilitating a high level of interoperability between, arguably, the two most important languages used in AI research: Python and Prolog, traditionally used for symbolic and sub-symbolic systems, respectively. We illustrate the `Rules2Lab` approach on a data-sharing case. We consider a scenario where vulnerabilities of a data-sharing infrastructure are explored by a reinforcement learning agent, and then conceptualized using inductive logic programming.

Paper structure In Section 2, we review related literature. n Section 3 we describe the main principles of `Rules2Lab`. At each step, definitions are exemplified on a data privacy use case. These examples build up the main components of a use case model later used in Section 4. With the use case model fully finalized, we illustrate the developed computational environment on experiments with deep reinforcement learning and norm induction. Finally, the results of simulation experiments are visualized and discussed. We conclude this paper in Section 5.

2 Background

This study is rooted in the question of first-order logic representation of a Markov decision process [29]. This research continues to be highly relevant following recent advances in relational reinforcement learning [25], extending to novel variants based on deep architectures [33, 54]. Languages tailored for logically reasoning agents in dynamic environments, that typically employ logic capable of capturing dynamic effects, have a long tradition [11], with GOLOG being one of its main representatives [35]. In contrast, `Rules2Lab` relaxes the need for temporal aspects, as state transitions are carried out separately in Python, making the environment relatively simpler. Since `Rules2Lab` places emphasis on ease of use and practical applicability, we present related literature broadly rather than venturing too deeply into specific matters.

2.1 Agent-based modelling

Agent-based environments are used to study emergent phenomena across several application areas [5,39]: card or computer games [48], city traffic [55], automated

[3] The code for `Rules2Lab` and illustrative examples are accessible at `https://github.com/fratric/Rules2Lab`

stock trading [37], or agents receiving feedback in natural language [13], to mention a few. These recent developments can be viewed as a special case of a more broad area of logic-based environments. Over the years, several languages, platforms, or environments were developed for multi-agent modelling. They differ mainly by their execution performance, user-friendliness of the language, learning abilities, or data integration [2]. Most of these are based on popular programming languages like Python or Java, with some platforms using their own dedicated language, such as AgentSpeak [40], NetLogo [49], or the SARL platform based on Janus language [28]. There seems to be a relatively lower level of development when it comes to cross-platform usage of logic-based technologies for MAS. NetProLogo can be regarded as most similar to our study.[4] In NetProLogo, the Netlogo multi-agent platform is extended with Prolog, e.g. to model the reasoning of agents [45]. In [15], Prolog is used to generate the environment, and a deep Q-learning-based agent in Python is used for exploration. With the SALMA platform [34], a domain model described by situation calculus and implemented in Prolog is extended with behavioral, domain-specific constructs implemented in Python. While these studies revolve around the question of how to best utilize Prolog and Python in one environment, none of them present a unified development pipeline. In contrast, our study aims to facilitate usage of state-of-the-art sub-symbolic libraries, such as RLlib [36] or TensorFlow [1], with a gym-like multi-agent environment that directly employs, or can be extended with, logic-based representations developed for multi-agent systems.

Starting from the point of view of sub-symbolic agents operating on top of symbolic structures, we can also invert the perspective by considering symbolic structures extracted from data generated by sub-symbolic agents. These are typically extracted into a form of behavioral norm. Various methods aim to identify and learn suitable logical expressions from behavioral data, e.g., grammatical versions of Markov chain Monte Carlo sampling [17], Bayesian approaches [16], or (probabilistic) inductive logic programming [46, 53]. Inductive logic programming has seen much progress in the development of new methods, going from traditional implementations such as Golem [38] or Metagol [19], to its cross-platform implementation Popper [20], and more recently implementations based on differentiable architectures such as ∂ILP [26] or DL-learner [10]. All these achievements are closely relevant to our environment-building approach, as it provides a flexible way to create new environments where such algorithms can be tested and benchmarked.

2.2 Data-sharing and norms

The question of behavioral norm-learning extends to the area of transfer learning, where the extracted knowledge can be shared between agents, or, more importantly to our case study, agents and humans [21]. Norms in AI are discussed in various contexts, most prominently: emergence of individual behavioral patterns [4], emergence of macro-regularities in a MAS [6], legal norms regulating

[4] Freely available online `https://github.com/jgalanp/NetProLogo`

agent behavior [52], or norms provided by designers aiming to align learning agents with human values [30]. This paper connects to the long-lasting research area of law formalization. Notably, Prolog or Prolog-like languages have been successfully used to model regulatory frameworks [8,14,41]. Following the enactment of the GDPR, scholarly attention within the realm of law formalization has notably gravitated towards the nuances and implications brought forth by this regulatory framework [3]. Several new languages to formalize the GDPR have been published [7,9,23,42]. Other research directions have developed languages extending to policy specification, typically targeting data security and compliance as their main application area, e.g., event-based Polder [22] or the eFlint language based on principles of deontic logic [50]. Both are rooted in logic-based programming. From an operational perspective, a common aspect of these studies is the presence of agents exchanging data on a virtual data marketplace [44].

3 Principles of `Rules2Lab`

`Rules2Lab` reinterprets Prolog knowledge bases as multi-agent systems, where *agents* take *actions* in an *environment*, advancing its *states*. The first step is to specify which predicates in the knowledge base define states, agents, actions, and observables.

Agents An agent is a constant term `x` specified by the predicate `isAgent(x)`. As an example, let us consider a data-sharing infrastructure with two agents:

```
isAgent(user).
isAgent(enforcer).
```

Agents in multi-agent systems typically take different actions depending on their type. The `user` agent is tasked with a standard interaction with the data-sharing infrastructure. The `enforcer` agent is tasked with a normative assessment of the state, and blacklisting non-compliant agents. Agents can be further categorized into various classes in order to specify which action agents may take, or what information is available to them. A trivial example of this categorization is `agent(X) :- user(X); enforcer(X)`, that can be used to query all agents in the environment, or to define whether an action can be taken by all or no agents.

States and observables The state of the environment is entirely recorded in the dynamically evolving Prolog knowledge base. However, from the perspective of agents, it can be useful to explicitly define which terms are relevant as a characteristic description of the environment state. For this reason, we explicitly wrap terms using the predicate `isStateVar(p,n)`, where `p` is a term of arity `n`.[5] This means that the state space S is defined by

– predicates $p_i, i = 1, ..., n$.

[5] `p` will later be used as a predicate, which the Prolog language allows.

– constant terms $t_i, i = 1, ..., m$.

Hence, a state $s \in S$ is expressed as a conjunction of atomic formulas, i.e., a set of predicates `p_i` grounded in a set of constant terms, such that `isStateVar(p_i, n_i)` holds. Effectively, s is represented as a database of true facts, and we refer to these atomic formulas as *state facts*. In this way, the state space admits a closed world assumption, i.e., every fact not explicitly stated as true is false.

As an example, let us consider 4 predicates of arity one, and one predicate of arity two, to define states of the environment:

```
isStateVar(user,1).
isStateVar(data,1).
isStateVar(private,1).
isStateVar(hasAccess,2).
```

These predicates make statements about object types present in the state space and their properties. In particular, the type `data` has properties defining if a data instance is private (`private`). Grounding these state variables with initial data, the state of the environment can be expressed as a conjunction of 5 state facts:[6]

```
user("insuranceCompany").
data("governmentData").
data("healthRiskData").
private("healthRiskData").
hasAccess("insuranceCompany", "governmentData").
```

If certain state variables are prohibited from being observed by every agent, or are simply not useful for the agent, then we can restrict observability of the environment. In order to define which state terms are observable for an agent `X`, we can write `isObservable(X,A,N)`, where for `A` it holds that `isStateVar(A,N)`. We define a set of observable states $O_x \subseteq S$, which are state variables that can be observed by the agent x.

Actions An action `a` is a constant term of arity `n` specified by the predicate `isAction(a,n)`. The set of actions is therefore defined by:

– action predicates $a_i, i = 1, ..., l$.
– constant terms $t_i, i = 1, ..., m$.

such that each $a \in A$ is expressed as a conjunction of ungrounded predicates consisting of ungrounded terms p_i and (built-in) arithmetic or logical relations. The agent x can then perform a mapping $\pi_x : O_x \to A$ using its policy function π_x, where A is a set of actions.

As an example, let us assume that the following norm must never be true in our knowledge base (See Figure 1a):

[6] We use Prolog `atoms` for terms that may be used as predicates, such as `user`, and Prolog `"strings"` for terms that represent entities, such as `"governmentData"`.

```
norm1(X,A) :- agent(X), data(A), hasAccess(X,A), private(A).
```

We can define action `isAction(gainAccess,2)` that respects the norm above. This mapping from a predicate present in the knowledge base into an action constitutes the central principle of `Rules2Lab`, and will be described in depth in Section 3.2.

3.1 State transitions

The state transition function performs change of state from state s and action a to the state s'. Formally, we define it as a mapping $\delta : S \times A \to S$. The state transition function can be viewed as a 'head' that reads the action and adds a new fact or rewrites an old fact. Although it is possible to utilize more advanced logic-based methods to model state transitions, certain actions might require complex statistical inference that simply does not have wide support in Prolog, while Python has a significant developer community creating supporting packages. Nonetheless, let us first define how a state transition can be implemented with only minimal Python involvement (symbolic case), and then discuss the implementation of state transitions where statistical inference might be needed (sub-symbolic case). In both cases, a program that can handle communication between Prolog and Python needs to be considered. We choose to use an implementation called `pyswip` that is written in Python [47].[7] Using this package, we implement a class called `handler`, that performs assertion and removal of state facts in the Prolog database.

Symbolic transitions In Prolog, we may specify changes in the state caused by `action` as `transition(action, state_facts, value)`, where `state_facts` is a list of grounded state variables that will be added or removed, and `value` is a value of this state transition. In the list `state_facts`, each state fact needs to be prefixed by an operator `+` or `-` to specify if the new fact becomes or ceases to be valid, respectively. Continuing in our example, we can write a state transition as:

```
transition(gainAccess(X,A), [+hasAccess(X,A))], 10).
```

where `X` is of type `user`, and `A` is of type `data`. The operator `+` specifies that the fact needs to be added into the collection of state facts. The last argument of the predicate states that the state transition is valued at 10.[8] A function implemented in Python then performs the state transition using the handler:

```
def step(self, action):
  t = self.handler.getTransition(action)
  for fact in t:
```

[7] More specifically, we use an implementation of `pyswip` that can isolate the communication if more than one channel is initiated: `https://github.com/mortacious/pyswip-notebook/blob/master/pyswip_notebook/prolog_notebook.py`.

[8] The value of the transition can be interpreted as a reward for agent `X`.

```
    if self.getOperator(fact) == '+':
      self.handler.addFact(fact)
    elif self.getOperator(fact) == '-':
      self.handler.removeFact(fact)
```

For brevity, we have omitted certain technical details. Moreover, the function typically returns information about state, action, or reward, in case we wish to perform further computations, e.g., reinforcement learning or data mining by analyzing trajectories.

Sub-symbolic transitions If no state transition is found in Prolog database, then the step function defaults to assuming that certain functionalities of Python are needed to advance the state. For example, consider an action `performInference(X,A,M)` taken by an agent `X` that uses a statistical model `M` to perform inference on data `A`. In this case, a state fact recording the information about inference being performed is added to the database along with new data. Following this example, we need to modify the step function as:

```
  if t is None:
    fact = self.groundStateVar('performedInference(X,A,M)',terms)
    self.handler.addFact(fact)
    performInference(X,A,M)
```

First, the handler records into the database that inference was performed. Second, a Python module is called to perform inference. Finally, the handler performs creation of a new data instance obtained from the inference module. We omit these details for brevity.

3.2 Action preconditions

Much like the state space, the action space can be built with a rich structure to specify conditions under which an agent may take an action. While the previous subsection dealt with transitions between states given an action, here, we specify how to define which actions are allowed in a given state. An action `a(X1,X2,...,Xk)` of arity k is actionable by the state transition function δ if and only if there exists a grounding of `X1, ..., Xk` into constant terms `t1, ..., tk` such that the conjunction `q1(t1,...,tk),q2(t1,...,tk),...,qn(t1,...,tk)` is true, where `q1`, `q2`, ..., `qn` are Prolog predicates. If the conditions in the conjunction are not satisfied, then the state transition δ cannot execute the action `a(t1,...,tk)`, hence agents are not allowed to take it. Trivially, an action that can be always taken by an agent is null action `isAction(nullAction, 1)`, i.e., an action not performing any change on the state of the environment. We may express its precondition as `nullAction(X) :- agent(X)`, which is always true.

Continuing with our running example, let us return to the `gainAccess` action of arity 2. We define this action to be actionable as follows:

```
gainAccess(X,A) :- user(X), data(A), not(private(A)),
                   not(hasAccess(X,A)).
```

The first two terms define that variable X must be of type user(X), and variable A must be of type data(A). The third term restricts actionability of this action to data that is not private. The last term is present to restrict redundancy of this action, i.e., to prevent an agent repeatedly requesting access to data after gaining it. Every grounding of X and A that satisfies these terms makes the action gainAccess(X,A) actionable, and does not violate consistency of our expert system knowledge base, i.e., norm1(X,A) is never evaluated as true.

Generation of action space In general, the actionability of an action depends on the state. However, it might be the case that an action depends on preconditions that have constant truth value throughout the simulation. If the action space is finite, then a separation of action preconditions into *static* and *dynamic* can be used to easily generate a finite vector of actions that are always actionable with respect to static facts. Clearly, the set of actionable actions with respect to static facts is always a superset of actions actionable with respect to both static and dynamic facts. This means we can obtain a minimal vector of all possible actionable actions simply by conveniently structuring the code. This functionality is implemented into the handler. If the handler is asked to generate a vector of possible actions, it will automatically search through all actions to find an action with the prefix static_. Let us illustrate this using our gainAccess(X,A) action. The action can be rewritten as:

```
static_gainAccess(X,A) :- user(X), data(A), not(private(A)).
gainAccess(X,A) :- static_gainAccess(X,A), not(hasAccess(X,A)),
```

Assuming that (1) there is a fixed number of users and data in the infrastructure, and (2) the privacy status of data cannot be changed, we can obtain all possible actions actionable independently of the state by querying the handler using the static version of gainAccess(X,A). Encodings of such vectors are very often used for deep learning architectures in reinforcement learning. As will be illustrated later, once the static terms are used to define the maximum length of the action vector, the dynamic terms can be used to decide which actions are actionable in the vector. This can be used to perform so-called *action masking* [32]. We deploy masking in combination with logic programming once we fully develop our case study model in Section 4. Note that a finite state vector can be generated analogously by splitting observables into static and dynamic components, as will be illustrated in Subsection 4.2.

3.3 Termination

An environment can have several terminating states for which the simulation terminates. Terminating states can be easily specified by formulating a predicate that defines under what conditions a state is a terminating state. Such a condition can be relatively trivial, e.g., no allowed actions are available to be taken by agents, making an environment truncate the simulation process. There can also be more sophisticated termination criteria defining conditions under which a hypothetical task was achieved by the agents.[9] To illustrate the more sophisticated terminating condition, we can define a termination predicate for our running example as:

```
terminate(X,A,B) :- user(X), data(A), data(B), private(A),
                    hasAccess(X,B), isIdentical(A,B).
```

This predicate states that if a `user(X)` gains access to a foreign data instance that is identical to a private data instance, then the simulation terminates. Clearly, this simple condition can be defined as a regulatory constraint into the environment, which means there is no added benefit to let agents solve this task. However, as discussed in Subsection 3.1, one can define a terminating state using functions performing statistical operations. As will be illustrated later, the exact identity condition `isIdentical(A,B)` in the example above can be relaxed with a predicate `isSimilar(A,B)`, evaluating an estimate of similarity compared to some threshold. This is very important, because the presence of terminating states defined using statistical methods can substantially complicate analysis of the underlying symbolic system using standard symbolic methods. Since verification of this constraint is not directly integrated into Prolog and ought to be implemented in Python, one can query the database with

```
terminate(X,A,B) :- user(X), data(A), data(B), private(A),
                    hasAccess(X,B).
```

and use the Python method to calculate similarity of each pair. Then the `enforcer` agent can blacklist the user `X`, or terminate the entire simulation, if necessary.

4 Experiment

Our target scenario — partially illustrated in Figure 1 — concerns identification of preconditions that lead to terminating states, defined as privacy breaches in a data marketplace. In turn, this will help identify new constraints to put into place to prevent agents from breaching privacy. We consider an expert system model of a data marketplace containing an inherent privacy vulnerability: a user can independently gain access to (1) personal data `"governmentData"` containing personal attributes and (2) a model `"healthRiskModel"` that predicts the health status of an individual. This means that health information about a specific person, which is a special category of personal data with enhanced protection under regulations such as GDPR (Art. 9), is inferred with some small degree of uncertainty. This vulnerability can be exposed by employing the `Rules2Lab` approach, and studying the behaviour of agents in a data-sharing laboratory environment. We consider a reinforcement learning agent called `"insuranceCompany"`, tasked with finding ways to obtain such information in the marketplace, thus exploiting its privacy vulnerabilities. The terminating states of the marketplace model are

[9] See distingtion between `is_done` and `truncate` terminating states used in Farama foundation *Gymnasium* standard `https://gymnasium.farama.org`.

defined by numeric similarity between private `"healthRiskData"` (See Table 1) and data inferred in the market from non-private sources freely available on the market.

Table 1: Sample of private `"healthRiskData"`. The first two columns are the same as publicly available `"governmentData"`, which means a model can be used to predict health risk from the age variable.

Name	**Age**	**Health Risk**
Alex Johnson	29	Low
Maria Gonzalez	34	Low
Kevin Smith	42	High
Emily Davis	27	Low
James Brown	31	Low

Once the expert system is mapped to a laboratory environment following the `Rules2Lab` approach, privacy vulnerabilities can be identified, and a new norm can be designed and implemented as a prohibitive access policy, making the market more compliant with legal rules, as in Figure 1d. The laboratory experiment used to perform this task is summarized as follows:

1. Let an agent motivated by reward learn a policy leading to terminating states of the environment.
2. Sample terminating and non-terminating trajectories.
3. Induce state conditions leading to terminating states.
4. Integrate new rules into the environment.
5. Perform evaluation of the environment with and without the new rules.

In the first step, we train an agent motivated by rewards (values of transitions) to discover trajectories that terminate the environment. In our running example, terminating states were identified with a privacy breach occurring in a data-sharing infrastructure. By designing the reward such that the terminating states yield high rewards, the reward-oriented agent can be viewed as an agent learning non-compliant behavior. In steps 2 and 3, we identify a predicate that describes a common property of all terminating trajectories, thus characterizing non-compliant behavior.[10] In step 4, we use this predicate to restrict agents' trajectories: in practice, the predicate is integrated as a new rule by adding new restrictions to the `gainAccess(X,A)` action. This means that `user` agents will need to satisfy more constraints before gaining access to a particular data or model. This process thus makes the infrastructure compliant by design, i.e., transforming a rule enforceable *ex-post* into a rule enforceable *ex-ante*. Finally, a single iteration might only capture some forms of non-compliant behavior. Further, the new constraint might induce other types of non-compliant behavior, or accidentally restrict too many trajectories and make the data-sharing infrastructure inoperable. Thus, in step 5, the environment needs to be evaluated with the new restrictions in place, returning to step 1 in an iterative process. We describe all components in more detail below.

[10] In general, characterizing non-compliant behavior is an active learning task [27].

4.1 Data sharing environment

In the simulation environment, we consider two types of user agents, differing by their policy function:

1. *Random user* that takes random actions with uniform probability.
2. *Reinforcement learning (RL) user* that learns an optimal policy using a proximal policy optimization algorithm [43].

We describe these agents in detail in the next sections. For now, let us define the basic functionalities of the data market model. In Table 2, we list state variables that define the state of the environment at each time step. Once the model is initialized with various users and data, the state can be visualized as a knowledge graph. See Figure 1a for entities loaded into the environment, adding one more user (`"advertismentCompany"`) and one more data (`"healthStatistics"`).

Table 2: State variables of the environment model in addition to basic `user`, `data`, and `model` types. Observable is market for the RL agent.

state fact	arity	argument types	RL agent
private	1	(data or model)	
hasPrivateAttributes	1	(data or model)	
hasMatchingVars	2	(data, model)	
hasAccess	2	(agent, data or model)	✓
performedInference	3	(agent, data, model)	

In addition to `gainAccess` action listed in Section 3, we add one more elementary action. The action allows a `user(X)` to create a new data instance by performing inference using a model `M` on data `A`. This action can be executed under the following constraints:

```
static_infer(X,A,B) :- user(X), data(A), model(B),
                       hasMatchingVars(A,B).
infer(X,A,B) :- static_infer(X,A,B), hasAccess(X,A),
                hasAccess(X,B), not(performedInference(X,A,B)).
```

The action `infer` needs to be associated with a method in Python that actually performs the inference, as discussed in Subsection 3.3. In our case, the inference is simply predictions drawn from the input model based on input data.

Lastly, terminating states are defined as discussed in Section 3. If a user intentionally or unintentionally obtains a data instance that is too similar to a private data instance, then this user is blacklisted by the `enforcer`. If all agents are blacklisted, then the simulation terminates, as there are no more agents left to take actions. The predicate `isSimilar(A,B)` compares similarity between data `A` and data `B`, and is true if the similarity is higher than threshold $\tau \in [0,1]$, false otherwise. We set $\tau = 0.90$. The similarity is calculated by comparing every row of private data `A` to inferred data `B` with matching column names, and recording the maximum number of identical elements. The maximum number for each row of `A` is recorded, and all values aggregated and divided by the size of number of entries of `A`.

A model execution that does not terminate is truncated, either due to only null actions being available for all agents, or if the maximum number of steps was reached.

4.2 Reward-based agent

We train deep neural network policies using proximal policy optimization, directly mapping observations from the set of observables O_x to the set of actions A. The RL agent is named `"insuranceCompany"`, and its vector of observables is constructed using

```
static_hasAccess("insuranceCompany", A) :- (data(A); model(A)),
    user("insuranceCompany").
```

where the user type variable is already grounded. We obtain the observation space as described in Section 3.2. First, by querying the static form of `hasAccess("insuranceCompany",A)` with respect to variable `A`, to obtain all possible data or models that the agent can have access to. Then, by representing this information as a boolean vector, input to the neural network, with `hasAccess("insuranceCompany",A)` specifying which boolean values are true or false. The output layer consists of all possible actions obtained as described in Subsection 3.2. Depending on dynamic state variables, the actions that the agent is not allowed to take are masked.

Data collection for the actor-critic based proximal policy optimization algorithm is performed by 7 agents running in parallel with a maximum of 20 steps. At every reset, each agent is initialized with the same initial state. The algorithm is implemented using the RLlib library [36]. The remaining hyperparameters, such as the network architecture, are set to the library's default values.

4.3 Norm induction

Several norm representations have been proposed in the literature [18]. While we are not using a dedicated language for norms in our illustrative multi-agent model, integration of a new norm follows guidelines formulated in [51]. If a predicate $P_a(x, s)$ about the state s becomes true, then the agent x is prohibited to take action a. In our case study, x is `user(X)` and a is `gainAccess(X,A)`, where `A` is data or a model. The predicate $P_a(x, s)$ is identified with a common property leading to terminating states, and needs to be induced from data sampled from the environment.

The environment can be sampled by executing agents' policy functions. At each step, each agent is sequentially allowed to take action. Each `user` generates a trajectory $(s_0, a_1, s_1, ..., a_n, s_n)$ labelled as positive or negative depending on whether the last action produced a terminating state, thus resulting in blacklisting of the `user` by the `enforcer`. To find the predicate $P_a(x, s)$ we use the inductive logic programming tool Popper [20]. This tool can construct predicates from positive and negative examples and background knowledge, even from noisy data. In our case, positive examples are identified with terminating trajectories, and negative examples with trajectories that did not terminate, meaning the trajectory was truncated. In practice, arity of the predicate needs to be specified in advance. We formulate our target predicate to be of arity 3. Furthermore, we assume that agents are not co-operative, hence we only need to consider state facts related to the user "insuranceCompany" as background knowledge. Both assumptions decrease the size of the search space. If no satisfying solution was found, the search space needs to be increased by relaxing the assumptions above.

4.4 Results

The RL agent was trained for 15 episodes with a maximum of 20 steps. The batch size was set to 640, $\gamma = 0.90$, and the learning rate 0.001. Average of total rewards per episode is visualized on Figure 2. After the training of the RL agent was completed, the multi-agent system was sampled to obtain 1000 trajectories for all three user agents operating in the model. Since the number of relations asserted during the simulation is only increasing, the end state of each trajectory contains the highest information value. We can therefore disregard previous states as redundant. To prevent agents from terminating the environment by brute force, we restrict the access rights to models and data to a maximum of two instances. After preparing the input for Popper, we run the inductive programming algorithm to perform induction over trajectories. We obtain the solution:

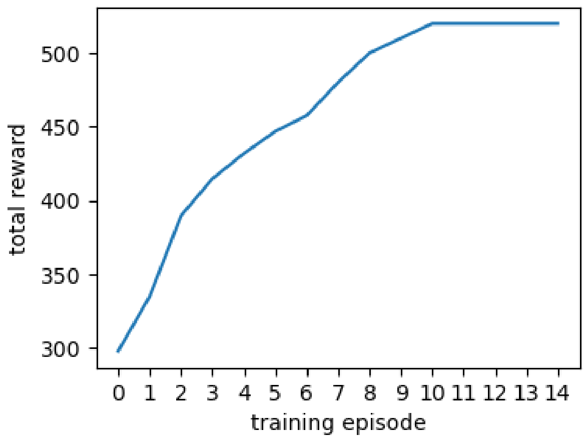


Fig. 2: Average of total rewards per episode of the RL agent ("insuranceCompany").

```
property(A,B,C):- performedInference(A,B,C),
    hasPrivateAttributes(B), hasPrivateAttributes(C).
```

which represents the common property of all terminating states.[11] We can use this property to finalize our prohibition norm as `P(X,property(X,B,C))`, which can be used to modify the access policy. After this dynamic modification, the amount of blacklisting by the `enforcer` drops to zero.

5 Conclusions

In this paper, we have introduced an agent-based modelling framework that delegates the majority of ontological specification to Prolog, and data-driven processing to Python. We demonstrated that the `Rules2Lab` approach can develop a logic-based gym-like environment connected to state-of-the-art sub-symbolic libraries, facilitating a high level of interoperability. Since developing novel environments from expert systems following the `Rules2Lab` approach requires only basic knowledge of Prolog and familiarity with gym environments commonly used for reinforcement learning, our approach is suitable for researchers and practitioners who seek to test their learning algorithms in easily extendable logic-based environments, or, alternatively, who seek to use sub-symbolic algorithms in their logic-based systems. The data privacy case study was chosen mainly to motivate and illustrate the model-building. Our approach can be used with small adjustments to any existing expert system implemented in, or compiled into, the Prolog language. Then, the power of contemporary research in sub-symbolic methods, many of which are implemented in the Python language, can be directly applied to test expert system, or conduct a variety of computational studies.

One must admit that the computational experiment conducted in this study is qualitative rather than quantitative. It is desirable to test our approach on a larger scale, and to explore conceptual improvements that may be needed, e.g., extending with more sophisticated types of logic, or with more powerful reinforcement learning algorithms. Nonetheless, based on our preliminary experience, there is reason to believe that our approach will scale well, as long as its individual components do so. Development of simulation environments where its components are viewed essentially as an input to a larger reasoning pipeline, potentially generated by a large language model as a foundation to a model-based reasoning agent, is the main focus of our future studies.

References

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al.: {TensorFlow}: a system for {Large-Scale} machine learning. In: 12th USENIX symposium on operating systems design and implementation (OSDI 16). pp. 265–283 (2016)

[11] Since generation of trajectories is stochastic, the algorithm might not convergence depending on the trajectory sample. This, however, is unlikely to happen for a large sample.

2. Ali, A.T., Leucker, M., Schuldei, A., Stellbrink, L., Sachenbacher, M.: A comparative analysis of multi-agent simulation platforms for energy and mobility management. In: Malvone, V., Murano, A. (eds.) Multi-Agent Systems. pp. 295–311. Springer Nature Switzerland, Cham (2023)
3. Amariles, D.R., Troussel, A.C., Hamdani, R.E.: Compliance generation for privacy documents under gdpr: A roadmap for implementing automation and machine learning. arXiv preprint arXiv:2012.12718 (2020)
4. Andrighetto, G., Villatoro, D., Conte, R.: Norm internalization in artificial societies. Ai Communications **23**(4), 325–339 (2010)
5. Antelmi, A., Cordasco, G., D'Ambrosio, G., De Vinco, D., Spagnuolo, C.: Experimenting with agent-based model simulation tools. Applied Sciences **13**(1), 13 (2022)
6. Balke, T., De Vos, M., Padget, J.: I-abm: combining institutional frameworks and agent-based modelling for the design of enforcement policies. Artificial Intelligence and Law **21**, 371–398 (2013)
7. Bartolini, C., Lenzini, G., Santos, C.: A legal validation of a formal representation of gdpr articles. In: Proceedings of the 2nd JURIX Workshop on Technologies for Regulatory Compliance (Terecom) (2018)
8. Bench-Capon, T.J.M., Robinson, G.O., Routen, T., Sergot, M.J.: Logic programming for large scale applications in law: A formalisation of supplementary benefit legislation. In: Proceedings of the First International Conference on Artificial Intelligence and Law, ICAIL '87, Boston, MA, USA, May 27-29, 1987. pp. 190–198. ACM (1987). `https://doi.org/10.1145/41735.41757`, `https://doi.org/10.1145/41735.41757`
9. Bhatia, J., Breaux, T.D.: Semantic incompleteness in privacy policy goals. In: 2018 IEEE 26th International Requirements Engineering Conference (RE). pp. 159–169. IEEE (2018)
10. Bühmann, L., Lehmann, J., Westphal, P.: Dl-learner—a framework for inductive learning on the semantic web. Journal of Web Semantics **39**, 15–24 (2016)
11. Calegari, R., Ciatto, G., Mascardi, V., Omicini, A.: Logic-based technologies for multi-agent systems: a systematic literature review. Autonomous Agents and Multi-Agent Systems **35**(1), 1 (2021)
12. Calegari, R., Ciatto, G., Omicini, A.: On the integration of symbolic and sub-symbolic techniques for xai: A survey. Intelligenza Artificiale **14**(1), 7–32 (2020)
13. Cheng, C.A., Kolobov, A., Misra, D., Nie, A., Swaminathan, A.: Llf-bench: Benchmark for interactive learning from language feedback. arXiv preprint arXiv:2312.06853 (2023)
14. Collenette, J., Atkinson, K., Bench-Capon, T.J.M.: Explainable AI tools for legal reasoning about cases: A study on the european court of human rights. Artif. Intell. **317**, 103861 (2023). `https://doi.org/10.1016/J.ARTINT.2023.103861`, `https://doi.org/10.1016/j.artint.2023.103861`
15. Costantini, S., De Gasperis, G., Migliarini, P.: Constraint-procedural logic generated environments for deep q-learning agent training and benchmarking. In: CILC. pp. 268–278 (2022)
16. Cranefield, S., Meneguzzi, F., Oren, N., Savarimuthu, B.T.R.: A bayesian approach to norm identification. ECAI 2016 (2016)
17. Cranefield, S., Savarimuthu, B.T.R.: Normative multi-agent systems and human-robot interaction. In: Workshop on Robot Behavior Adaptation to Human Social Norms (TSAR) (2021)
18. Criado, N., Argente, E., Botti, V.: Open issues for normative multi-agent systems. AI communications **24**(3), 233–264 (2011)

19. Cropper, A., Dumančić, S.: Inductive logic programming at 30: a new introduction. Journal of Artificial Intelligence Research **74**, 765–850 (2022)
20. Cropper, A., Morel, R.: Learning programs by learning from failures. Machine Learning **110**(4), 801–856 (2021)
21. Da Silva, F.L., Costa, A.H.R.: A survey on transfer learning for multiagent reinforcement learning systems. Journal of Artificial Intelligence Research **64**, 645–703 (2019)
22. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The ponder policy specification language. In: Sloman, M., Lupu, E.C., Lobo, J. (eds.) Policies for Distributed Systems and Networks. pp. 18–38. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
23. De Vos, M., Kirrane, S., Padget, J., Satoh, K.: Odrl policy modelling and compliance checking. In: Fodor, P., Montali, M., Calvanese, D., Roman, D. (eds.) Rules and Reasoning. pp. 36–51. Springer International Publishing, Cham (2019)
24. Dulac-Arnold, G., Levine, N., Mankowitz, D.J., Li, J., Paduraru, C., Gowal, S., Hester, T.: Challenges of real-world reinforcement learning: definitions, benchmarks and analysis. Machine Learning **110**(9), 2419–2468 (2021)
25. Džeroski, S., De Raedt, L., Driessens, K.: Relational reinforcement learning. Machine learning **43**, 7–52 (2001)
26. Evans, R., Grefenstette, E.: Learning explanatory rules from noisy data. Journal of Artificial Intelligence Research **61**, 1–64 (2018)
27. Fratrič, P., Parizi, M.M., Sileno, G., van Engers, T., Klous, S.: Do agents dream of abiding by the rules? learning norms via behavioral exploration and sparse human supervision. In: Proceedings of the Nineteenth International Conference on Artificial Intelligence and Law. pp. 81–90 (2023)
28. Galland, S., Rodriguez, S., Gaud, N.: Run-time environment for the sarl agent-programming language: the example of the janus platform. Future Generation Computer Systems **107**, 1105–1115 (2020). `https://doi.org/https://doi.org/10.1016/j.future.2017.10.020`, `https://www.sciencedirect.com/science/article/pii/S0167739X17313419`
29. Guestrin, C., Koller, D., Gearhart, C., Kanodia, N.: Generalizing plans to new environments in relational mdps. In: Proceedings of the 18th international joint conference on Artificial intelligence. pp. 1003–1010 (2003)
30. Hadfield-Menell, D., Russell, S.J., Abbeel, P., Dragan, A.: Cooperative inverse reinforcement learning. Advances in neural information processing systems **29** (2016)
31. Hamdani, R.E., Mustapha, M., Amariles, D.R., Troussel, A., Meeùs, S., Krasnashchok, K.: A combined rule-based and machine learning approach for automated gdpr compliance checking. In: Proceedings of the Eighteenth International Conference on Artificial Intelligence and Law. p. 40–49. ICAIL '21, Association for Computing Machinery, New York, NY, USA (2021). `https://doi.org/10.1145/3462757.3466081`, `https://doi.org/10.1145/3462757.3466081`
32. Huang, S., Ontañón, S.: A closer look at invalid action masking in policy gradient algorithms. arXiv preprint arXiv:2006.14171 (2020)
33. Jiang, Z., Luo, S.: Neural logic reinforcement learning. In: International conference on machine learning. pp. 3110–3119. PMLR (2019)
34. Kroiß, C., Bureš, T.: Logic-based modeling of information transfer in cyber–physical multi-agent systems. Future Generation Computer Systems **56**, 124–139 (2016). `https://doi.org/https://doi.org/10.1016/j.future.2015.09.013`, `https://www.sciencedirect.com/science/article/pii/S0167739X15002939`

35. Levesque, H.J., Reiter, R., Lespérance, Y., Lin, F., Scherl, R.B.: Golog: A logic programming language for dynamic domains. The Journal of Logic Programming **31**(1-3), 59–83 (1997)
36. Liang, E., Liaw, R., Nishihara, R., Moritz, P., Fox, R., Goldberg, K., Gonzalez, J., Jordan, M., Stoica, I.: Rllib: Abstractions for distributed reinforcement learning. In: International conference on machine learning. pp. 3053–3062. PMLR (2018)
37. Liu, X.Y., Yang, H., Chen, Q., Zhang, R., Yang, L., Xiao, B., Wang, C.D.: Finrl: A deep reinforcement learning library for automated stock trading in quantitative finance. arXiv preprint arXiv:2011.09607 (2020)
38. Muggleton, S.H., Feng, C., et al.: Efficient induction of logic programs. Turing Institute (1990)
39. Railsback, S.F., Lytinen, S.L., Jackson, S.K.: Agent-based simulation platforms: Review and development recommendations. Simulation **82**(9), 609–623 (2006)
40. Rao, A.S.: Agentspeak (l): Bdi agents speak out in a logical computable language. In: European workshop on modelling autonomous agents in a multi-agent world. pp. 42–55. Springer (1996)
41. Satoh, K.: PROLEG: practical legal reasoning system. In: Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M.V., Kowalski, R.A., Rossi, F. (eds.) Prolog: The Next 50 Years, Lecture Notes in Computer Science, vol. 13900, pp. 277–283. Springer (2023). `https://doi.org/10.1007/978-3-031-35254-6_23`, `https://doi.org/10.1007/978-3-031-35254-6_23`
42. Sawasaki, T., Satoh, K., Troussel, A.C.: A use case on gdpr of modular-proleg for private international law. In: AI4LEGAL/KGSum@ ISWC. pp. 1–11 (2022)
43. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017)
44. Shakeri, S., Maccatrozzo, V., Veen, L., Bakhshi, R., Gommans, L., De Laat, C., Grosso, P.: Modeling and matching digital data marketplace policies. In: 2019 15th International Conference on eScience (eScience). pp. 570–577. IEEE (2019)
45. Stamatopoulou, I., Sakellariou, I., Kefalas, P.: Formal agent-based modelling and simulation of crowd behaviour in emergency evacuation plans. In: 2012 IEEE 24th International Conference on Tools with Artificial Intelligence. vol. 1, pp. 1133–1138 (2012). `https://doi.org/10.1109/ICTAI.2012.161`
46. Tan, Z.X., Brawer, J., Scassellati, B.: That's mine! learning ownership relations and norms for robots. In: Proceedings of the AAAI conference on artificial intelligence. vol. 33, pp. 8058–8065 (2019)
47. Tekol, Y., contributors: PySwip v0.2.10 (2020), `https://github.com/yuce/pyswip`
48. Terry, J., Black, B., Grammel, N., Jayakumar, M., Hari, A., Sullivan, R., Santos, L.S., Dieffendahl, C., Horsch, C., Perez-Vicente, R., et al.: Pettingzoo: Gym for multi-agent reinforcement learning. Advances in Neural Information Processing Systems **34**, 15032–15043 (2021)
49. Tisue, S., Wilensky, U.: Netlogo: A simple environment for modeling complexity. In: International conference on complex systems. vol. 21, pp. 16–21. Citeseer (2004)
50. Van Binsbergen, L.T., Liu, L.C., Van Doesburg, R., Van Engers, T.: eflint: a domain-specific language for executable norm specifications. In: Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences. pp. 124–136 (2020)
51. Vázquez-Salceda, J., Aldewereld, H., Dignum, F.: Implementing norms in multi-agent systems. In: Lindemann, G., Denzinger, J., Timm, I.J., Unland, R. (eds.) Multiagent System Technologies. pp. 313–327. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)

52. Williams, T.G., Brown, D.G., Guikema, S.D., Logan, T.M., Magliocca, N.R., Müller, B., Steger, C.E.: Integrating equity considerations into agent-based modeling: a conceptual framework and practical guidance. Journal of Artificial Societies and Social Simulation **25**(3) (2022)
53. Xu, D., Fekri, F.: Interpretable model-based hierarchical reinforcement learning using inductive logic programming. arXiv preprint arXiv:2106.11417 (2021)
54. Zambaldi, V., Raposo, D., Santoro, A., Bapst, V., Li, Y., Babuschkin, I., Tuyls, K., Reichert, D., Lillicrap, T., Lockhart, E., et al.: Relational deep reinforcement learning. arXiv preprint arXiv:1806.01830 (2018)
55. Zhang, H., Feng, S., Liu, C., Ding, Y., Zhu, Y., Zhou, Z., Zhang, W., Yu, Y., Jin, H., Li, Z.: Cityflow: A multi-agent reinforcement learning environment for large scale city traffic scenario. In: The world wide web conference. pp. 3620–3624 (2019)