The Secret Life of Traces: a MAS Engineering Perspective

Davide Ancona[1[0000−0002−6297−2011]], Zahra Daoui[1[0009−0000−7217−2010]], Angelo Ferrando[2[0000−0002−8711−4670]], Andrea Gatti[1[0009−0003−0992−4058]], Viviana Mascardi[1[0000−0002−2261−9926]], and Michael Winikoff[3[0000−0002−5545−7003]]

[1] University of Genova, Italy, davide.ancona@unige.it, zahra.daoui@edu.unige.it, andrea.gatti@edu.unige.it, viviana.mascardi@unige.it
[2] University of Modena-Reggio Emilia, Italy, angelo.ferrando@unimore.it
[3] Victoria University of Wellington, New Zealand, michael.winikoff@vuw.ac.nz

Abstract. In this paper we explore the secret life of traces. In particular (i) we review the scientific literature on execution traces, with a special attention to their exploitation in the multiagent systems area; (ii) we propose a unified theoretical architecture, NOVAE, based on the surveyed literature and on our experience with traces in the runtime verification, explainability and learning domains; and (iii) we suggest – by means of examples implemented on top of VEsNA– how to exploit traces for the three problems above. VEsNA is a toolkit that integrates the Jason interpreter for the AgentSpeak(L) language, the Godot and Unity game engines, and natural language interfaces. It was selected as a representative of complex systems integrating symbolic agents.

Keywords: Traces; Multiagent System; Runtime Verification; Explainability; Learning; Neuro-symbolic agents; Verify, Learn and Explain; NOVAE

1 Introduction

Execution traces, broadly understood as recorded sequences of events or state transitions – along with a representation of the states themselves, in some contexts – produced by a running system, are a foundational concept that pervades many areas of computer science. In *software engineering*, traces are the primary input for dynamic program analysis [24], automated log analysis [53,21,60], and specification mining [3]. In *formal methods*, counterexample traces are among the most valuable outputs of model checkers [30], and runtime verification evaluates traces against formal specifications during execution [12,62,80]. In *machine learning*, traces appear as trajectories from which reinforcement learning agents derive policies [84], as expert demonstrations for imitation learning [44], and as the sequences over which decision transformers perform sequence modeling [22]. In *distributed systems*, distributed traces record the end-to-end flow of requests across microservices and constitute one of the three pillars of observability [14,63].

Although traces are used everywhere, the notion of what constitutes a trace varies considerably across domains [74]. Addressing the role of traces in a unified way is therefore hard for at least three reasons:

Terminological: what is named 'trace' in the runtime verification (RV) area is named 'event sequence' in the machine learning and in the data visualization areas, and 'event log' in the process mining one; 'traceability' has a very specific meaning in software engineering [82] creating further noise around the word 'trace' as we use it.

Foundational: depending on the foreseen use of the trace, some features may or may not be mandatory in the trace elements; for example, according to [87] '*an event sequence is a sequence of timestamped observations, each described by a fixed set of features*'. This definition requires that the elements of the trace are timestamped events, whereas in other domains including RV the ordering of events must be tracked but not necessarily their timestamp. When traces are used for explainability purposes, not only observable events (for example, 'going_to_restaurant') are needed in order to explain the behavior of an agent, but also some representation of the internal (for example, 'feeling_hungry') and/or external (for example, 'empty_fridge') states where the event took place.

Representational: since there is no unique set of features that an element of a trace should support, there is no standard representation for such elements, and hence no standard representation for traces. This is true both for the representation of a single trace and for the compact and efficiently usable representation of a set of traces. As an example, just in the RV domain, the Linear Time Temporal Logic language [70] commonly used to succinctly denote sets of traces by means of logical properties, comes with dozen variants and extensions [80]. Non logic-based languages also exist [7], again with their variants [5,6,23].

What remains constant despite the above challenges is the underlying purpose of traces that provide an inspectable, online or offline abstraction of what a system did, in what order, and under what conditions.

The objective of this paper is to design a theoretical unified architecture for **N**eurosymb**O**lic agents that **V**erify, le**A**rn and **E**xplain (**NOVAE**), based on the notions of *trace element*, *trace* (a sequence of such elements), *trace logger* (a software tool that is integrated with, or that monitors, a system in order to produce a trace). These notions must be as general as possible, to fit the features of traces as we find them in the RV, learning, and explainability domains. Still, they must be concrete and practical enough to design algorithms working on traces. Our long-term goal, in fact, is to implement a NOVAE instance on top of the VEsNA toolkit [39,48], an ecosystem of software tools for cognitive software agents situated in virtual environments and able to interact using natural language.

The NOVAE architecture is described in Section 3, and its conception is firmly grounded in the scientific literature described in Section 2. The path towards implementing a NOVAE instance in VEsNA is presented in Section 4 via examples that are implemented – albeit not coherently integrated into a 'NOVAE VEsNAe' yet. Section 5 concludes.

2 Fantastic Traces and where to Find them

Runtime Verification. '*Runtime [...] Verification deals with traces*'. This is how the abstract of the paper 'What is a Trace? A Runtime Verification Perspective' by Reger

and Havelund [74] starts. Runtime verification is a lightweight formal verification technique that monitors the observable execution of a system (not its internal state evolution) against formal specifications to dynamically detect violations of desired properties, instead of verifying all possible executions statically [36,32]. Unlike model checking, which exhaustively explores state spaces offline, RV focuses on observing actual system behavior during execution.

In MAS, monitors check events belonging to execution traces to ensure that agents satisfy safety properties, goal achievement constraints, and behavioral specifications during deployment [4]. RV4JaCa [31] provides runtime verification for JaCaMo-based systems, allowing developers to specify temporal properties over agent behaviors and environment interactions. For decentralized systems, monitoring approaches distribute verification across agents, maintaining local monitors that coordinate to verify global properties [33]. Assumption violation recognition [35] models environmental assumptions and monitors their violations, enabling adaptive responses when operational contexts change. Beyond these approaches, other RV frameworks for MASs have been proposed. Runtime verification of MAS interaction quality [10], runtime verification frameworks for dynamically adaptive multi-agent systems [64], and (semi-)runtime approaches for formal verification of JADE-based MASs [78,18] each address different facets of the problem. The combination of model checking and runtime verification for strategic reasoning under imperfect information [40] and safety shields [34] that constrain BDI (Beliefs-Desire-Intentions [73]) plan execution at runtime to prevent safety violations further illustrate the breadth of RV applications in MAS.

Explainability. The ability of a system to explain itself is important for a range of reasons (e.g. transparency [88,58,54], calibrating trust [61,89,75], acceptability [42], understandability [86], accountability [25], and traceability [86]). There is consequently a whole body of work on eXplainable AI (XAI) [52,8]. However, much of this work concerns explanations of machine learning rather than of autonomous systems (respectively termed 'data-driven XAI' and 'goal-driven XAI' [8] or 'explainable agency' [61]).

Explaining the behavior of an autonomous system in essence involves answering questions of the form 'why did you do X?', where X is an action that was performed by the autonomous system, and that occurs in a trace. The trace is an essential input to the explanation generation process, since explanations can include action pre-conditions and since explaining why a given plan was selected may require information on the state (e.g. the agent's beliefs) at a given point in time. In addition to a trace, the explanation generation process also uses the agent's plan library. The trace and plan library together allow explanations to be generated in terms of human-oriented concepts such as goals, beliefs, and valuings [95]. Recent work includes: defining a *scoresheet* [96] to capture the explainability features of a given system (or the requirements for explainability of a system-to-be); extending explanations back to the *requirements* of a MAS [76,77]; proposing an *architecture* for explainable multi-agent system upon which explainability can be built [90]; extending it to handle *contrastive* questions ('why did you do X rather than Y?') [91]; and conducting empirical evaluations of explanation [93,94,91]. Beyond goal-driven explainability for BDI agents, other dimensions of explainability in MAS have been explored. Explanations need not be limited to human-directed communication: agent-to-agent explanation, where agents provide justifications to one an-

other to facilitate coordination, negotiation, and trust-building [68], is equally relevant in open MAS. The integration of symbolic and sub-symbolic techniques for explainable AI [20] further highlights how hybrid neuro-symbolic approaches can leverage symbolic reasoning (as found in BDI agents) to provide interpretable explanations of behaviors that may involve learned components. Accountability in multi-agent organizations [11,67] offers yet another perspective, showing how traces of agent interactions within organizational structures can be used to attribute responsibility for outcomes, shifting the focus from 'why was this done' to 'who is responsible'. Across all these perspectives, execution traces remain the evidential foundation, recording what happened, in what order, and under what conditions, and providing the raw material for building explanations.

Learning. Traces play a fundamental role in machine learning, where they are often referred to as *trajectories*, *episodes*, or *experience sequences*. In this context, a trace captures the temporal evolution of a system's interaction with its environment, providing the raw data from which learning algorithms extract patterns, policies, or predictive models. In Reinforcement Learning (RL), the agent learns optimal behavior through interaction with an environment, generating traces of state-action-reward sequences [84]. The foundational concept of *eligibility traces* directly embodies the role of execution history in learning: eligibility traces maintain a record of recently visited states, enabling credit assignment to be propagated backward through the trajectory [84]. Deep Reinforcement Learning (DRL for short) extends this approach by introducing *experience replay* [65], where traces are stored in a replay buffer and are sampled during training. This technique breaks the temporal correlation between consecutive experiences, stabilizing learning in neural network-based approaches. The stored traces – tuples of (s_t, a_t, r_t, s_{t+1}) where s_i are states, a_i actions, and r_i rewards – constitute explicit execution histories that are repeatedly revisited to refine the learned policy. The success of Deep Q-Networks [65] demonstrated that maintaining and learning from large collections of traces enables agents to achieve human-level performance in complex domains. Beyond reinforcement learning, traces underpin both *imitation learning* and *learning from demonstrations*: behavioral cloning learns a direct state-to-action mapping from expert traces [44], while inverse reinforcement learning recovers the reward function that explains the observed behavior, showing that traces encode not only what happened but also the rationale behind decisions. More recent sequence-based approaches such as *decision transformers* [22] reframe reinforcement learning as sequence modeling over traces of states, actions, and returns. When safety is a concern, *shielding* techniques [2] constrain exploration so that generated traces never violate safety specifications [44]. In *process mining* [1], knowledge is extracted from event logs recorded during system execution. In MAS settings, process mining has been used to analyze agent activity logs and reconstruct the interaction patterns that emerge from autonomous behavior [79,13].

In MAS, learning from traces presents unique challenges including partial observability, non-stationarity (as other agents learn simultaneously), and the need for coordination. Traces generated by individual agents capture only local perspectives, raising questions about how to aggregate or share experiential data across the system. The integration of learning capabilities into BDI agents combines symbolic reasoning transparency with machine learning adaptability. Early work explored reinforcement learn-

ing for plan selection [81], followed by embeddings of various RL algorithms within BDI reasoning cycles, including Temporal Difference Learning [9], SARSA-based selection [17], and Deep RL approaches [69].

3 NOVAE Vision: Neuro-symbOlic agents that Verify, leArn and Explain

In this section we put forward our vision for **N**euro-symb**O**lic agents that **V**erify, le**A**rn and **E**xplain, **NOVAE**. Our vision applies to a *System Under Analysis*, a SUA[4], that produces traces thanks to a *Trace Logger*. Depending on the components of the trace elements, this logger might need to have access to the internal state of the SUA, or to be aware of its surrounding environment, or it might just observe events (a light that turns from blue to red, a movement of a robot in the real environment, an action of an agents in the virtual one, a sentence printed on a user interface). In order to be general enough and deal with these different trace elements in a coherent and unified way, our *Trace Elements* are pairs of *States* and *Changers*, as defined in Section 3.1. Either the state or the changer might be missing. This vision is more general than the most common one, where the trace is 'a sequence of events'. It easily accommodates timestamped events, with timestamp belonging to the state and the observed event being the changer, and complex trace elements where internal/external state representation must also be recorded.

The vision is also general enough to be applicable to any kind of SUA that may be instrumented to capture execution traces via the *Trace Logger*, no matter its underlying architecture and implementation technology. In our past and recent research activities we already addressed the *verify* and *explain* problems in relation with MAS and with BDI (and Jason, in particular [16]) agents – see the already mentioned works [4,5,18,32,33,35,36] and [76,77,90,91,93,94,95,96], respectively. Sections 3.2 and 3.3 adapt the NOVAE vision to verification and explainability, respectively, taking our previous work into account. The *learning* problem, instead, is quite new for us and it is worth being addressed in more detail, in particular with respect to the role of traces in the learning process. Section 3.4 is devoted to dealing with this challenge. We emphasize the neuro-symbolic flavor of NOVAE because the NOVAE instantiation we are developing, presented in Section 4, integrates symbolic and neural aspects. This raises challenges but also exciting opportunities. NOVAE itself is however agnostic of the SUA implementation.

3.1 NOVAE Traces

Our definition of a NOVAE trace applies to a SUA that produces traces, and relies on the notions of *State* and *Changer*. As anticipated, traces are captured by a *Trace Logger*. How the SUA may be instrumented to produce those traces is outside the scope of our investigation. Many tools are available to support this stage [21].

[4] SUA is inspired by SUS, *System Under Scrutiny*, which is a standard terminology in the RV research area. Systems in the NOVAE vision are analyzed for verifying and explaining their current behavior, and for learning new behaviors, hence the SUA acronym.

Changers change the SUA's execution state. They may be though of as the 'events' mentioned in the trace-related literature, but we want to stress their active and effective nature. Changers may also change the SUA's environment, but the effects of changers on the environment is not directly recorded within traces: the target of our monitoring/-explanation/learning activity is in fact the SUA, not its environment.

We make no assumptions on how the SUA state looks like: in the sequel, we provide examples that apply to SUAs that are symbolic frameworks, where states may include the epistemic knowledge base (facts, beliefs) and the procedural knowledge base (rules, plans), and to SUAs that are black boxes. We refrain from using the term 'action' to identify elements of the SUA that cause changes to its state – that we name in fact 'changers' – because in our target framework, VEsNA, 'action' has a specific technical meaning and VEsNA actions are a subset of VEsNA changers. Also, more generally, in a MAS 'actions' affect the environment, whereas a changer might affect the SUA internal state only, for example via deliberation, planning, reasoning.

The SUA execution starts because a *Run Event* takes place. While this run event is a changer in most cases, other events may trigger the SUA execution and we introduce this distinction to keep our definition as general as possible.
A NOVAE trace is a triple

$\langle$*Run Event*, *State-Changer Sequence*, *Final State*$\rangle$

where the State-Changer Sequence is a sequence of pairs

[*(State*$_0$*, Changer*$_0$*), (State*$_1$*, Changer*$_1$*), ..., (State*$_n$*, Changer*$_n$*)*]

The *Final State* is the state of the SUA when its execution stops – whatever the reason for the interruption. We do not represent it inside the State-Changer Sequence, because there is no changer associated with the final state, and hence no pair *(Final State, Changer*$_{Final\ State}$*)*.
The State-Changer Sequence is determined by a SUA-dependent relation $\mathcal{D} \subset ((STATE \times CHANGER) \times \mathbb{P}(STATE))$ where $STATE$ is the set of states, $CHANGER$ is the set of changers, and *((State*$_n$*, Changer*$_n$*), SET*$_{n+1}$*)* $\in \mathcal{D}$ means that the execution of *Changer*$_n$ in *State*$_n$ leads to one among many possible states in a set SET$_{n+1}$; [*(State*$_n$*, Changer*$_n$*), (State*$_{n+1}$*, _)*] is a sub-sequence of the State-Changer Sequence iff *((State*$_n$*, Changer*$_n$*), SET*$_{n+1}$*)* $\in \mathcal{D}$, and *State*$_{n+1} \in$ *SET*$_{n+1}$.

SUA	Program	State	Run Event	Changer
An IoT Device	Javascript or Python code	The values of variables that characterize the state of the device (temperature, pressure, light, ...), the current time, some internal data related with the device as its current CPU usage	Switch the IoT device on	Actions taken by the device based on its current state (send a signal to a controller or to another device)
Chain of Thought in a Large Language Model	The Deep Neural Network that constitutes the Large Language Model (LLM)	The state of the LLM, or a subset of it; despite being extremely complex and not inspectable, the LLM has a state – as any other advanced software module – represented by its neurons and parameters	The initial prompt issued to the LLM	The prompt that the LLM issues to itself, as part of the Chain of Thought process

Table 1. Examples of non-symbolic SUAs.

Traces like these may be produced by any SUA as shown by the two examples in Table 1, that represent two quite different applications, one from the Internet of Things

(IoT) domain and one related with an LLM Chain of Thought. If we restrict our investigation to symbolic SUAs, their program may be abstracted into a Knowledge Base, *KB*, and a *set of rules* – where rules mean, in a very general way, declarative specifications of the SUA behavior characterized by applicability conditions that depend on the KB, and a body defining what to do when such conditions are met. The KB and the rules may contribute to the SUA state, along with other run-time components, operational information, and timestamps. Depending on the problem at hand, the SUA state might be a sub-symbolic embedding of declarative components into a vector space, paving the way to efficient processing for learning purposes. SUAs in Table 2 are characterized by some common features – KB and Rules – due to their symbolic nature.

SUA	KB	Rules	State	Run Event	Changer
A Prolog Module [83]	Set of Prolog facts	Set of Prolog clauses, where the applicability condition is represented by the head of the rule, and the body by the sequence of goals in the rule body	Execution stack recorded for backtracking purposes, current set of facts and/or clauses	A goal call (that may be a conjunction of goals) provided to the interpreter	A call to an atomic goal
A STRIPS planner program [41]	Initially true conditions	Actions characterized by their pre- and post-conditions; the applicability condition is the pre-condition	Conditions that are currently true	The goal state provided to the planner	STRIPS actions
An AgentSpeak(L) agent [72]	Belief base	AgentSpeak(L) plans that are applicable when an event matching the plan's triggering event took place, and the plan's context is a logical consequence of the current KB	The current beliefs and plans, and the current state of the intentions, event queue, message queue	A set of initial achievement goals	Internal and user-defined actions, actions on the environment, updates of beliefs, test and achievement goals

Table 2. Examples of symbolic SUAs.

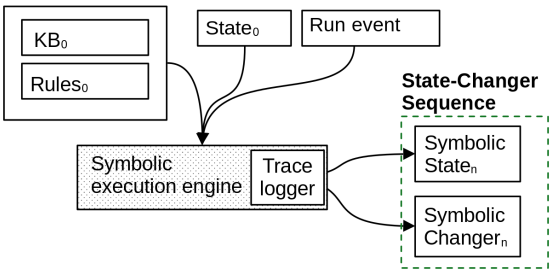

Fig. 1. NOVAE execution with trace logging.

Examples in Tables 1 and 2 show that our trace definition is general, and many SUAs are compliant with it. Figure 1 shows a SUA – more precisely a symbolic agent implemented via a KB and a set of rules – whose execution engine is instrumented with a trace logger. A State-Changer Sequence is produced by the logger. In the next three

sections we discuss how this architecture may suit the agent's verification, explainability and learning needs, thanks to addition of modules on top of its core components.

3.2 NOVAE Verification

The NOVAE vision easily fits the typical RV architecture, as shown in Figure 2: since RV verifies that observed events meet the specification of the expected behavior, there is no need for the monitor (the RV engine) to take states into account. Hence, the trace relevant for RV purposes consists only of changers. The emitted verdict is based on a three-valued logic to take into account inconclusive cases due to missing information in the trace.

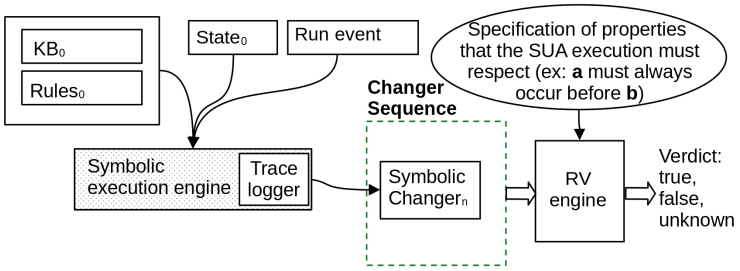

Fig. 2. NOVAE **runtime verification**: traces consisting of changers feed a monitor (also named RV engine) that emits a verdict on changers' compliance w.r.t. the specification (see [7], Fig. 1).

3.3 NOVAE Explainability

Figure 3 re-interprets the architecture presented in [90] in terms of traces. This re-interpretation is a very natural one, since the explainability infrastructure described therein assumed the presence of 'Black Boxes' *that capture relevant details from the system's execution*, which are exactly what we name 'Trace Logger' in Figure 1.

Quoting [90] again, '*to explain a component that uses rule-based reasoning, the black box would likely need to capture the facts that were believed to be true at a given point in time that were the basis for the choice of rule that was made*'. This is coherent with the expected outcome of the trace logger that logs states and changers, where states might consist of a subset of the agent's beliefs/facts. Users (on the right) interact with an interface that maintains a model of them to filter out answers. When users ask questions, the interface passes them to the explainer, that might consist of many different customized explainers, among which the most suitable one should be selected. The explainer needs to know what happened in the past (changers), and under which conditions (states), in order to answer 'Why' and 'Why not' questions, together with 'What if' ones.

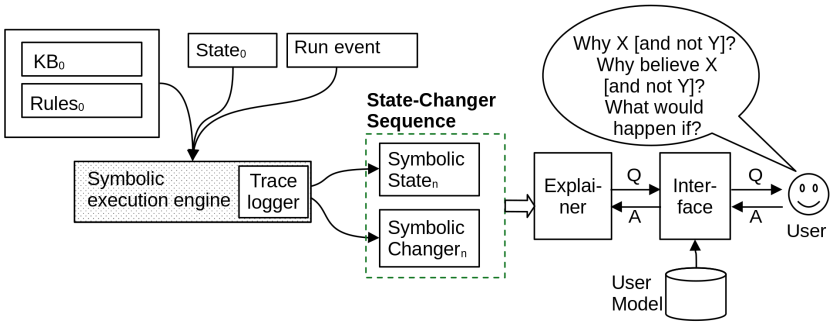

Fig. 3. NOVAE **explainability**: an interface that takes the user's model as input lies in between the user asking for explanations and the explainer (see [90], Fig. 1).

3.4 NOVAE Learning

The objective of this section is to answer the question: how could a symbolic agent described by its KB and rules learn new appropriate courses of changers that, from an initial state, lead to one among a given set of final states?

We envision two approaches; one where *explicit* traces are not needed and learning is delegated to an external learning module ('(Deep) Reinforcement Learning as a Service'), and another based on explicit traces generated by randomly creating KBs and rules, inspired by genetic algorithms [55]. The important aspect here is that the symbolic execution system is run many times under different conditions, with some randomness. Traces are generated even if changers and rules are random, but most traces will not lead to an acceptable final state and will be marked as failures. Still, some traces will successfully lead to a final state: the initial KB, rules, and run event that by chance lead to a final state might contribute to learning how to reach that final state, in a creative way. All the annotated traces will be supplied to a 'Rule and KB Learner' tool, that will learn rules that lead to successful traces along with the conditions (the KB) that make these rules applicable.

(Deep) Reinforcement Learning as a Service. Figure 4 shows our vision, where 'as a Service' emphasizes that the learning algorithm is not inside the agent (programmed in the same programming language as the agent is) but outside it.

As observed in Section 2, the exploration stage of (Deep) Reinforcement Learning (Figure 4(a)) is based on online traces that meet our general definition: the run event takes place on the sub-symbolic side of the picture and consists of initializing the learning agent with the (properly encoded) initial state and the available changers, plus a 'start learning' command; at every iteration within a learning epoch, a state (including the reward) is sent to the learning engine, and a changer is output by it. In this setting, the trace consists of states only, since changers are not produced by the symbolic execution engine, but by the (Deep) Reinforcement Learning engine. This represents the opposite with respect to the RV setting, where traces consist of changers only. The final state is the state of the agent after either the goal or the expected number of iterations is

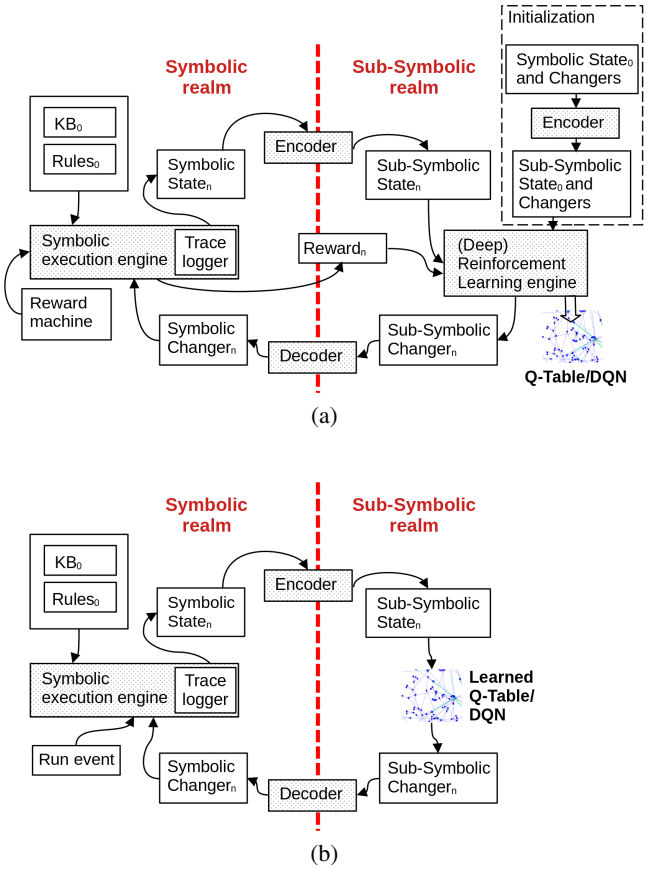

Fig. 4. (Deep) Reinforcement Learning as a Service: (a) **exploration**, the agent is enriched with a reward machine; its states are encoded and fed to a (D)RL engine, properly initialized; the (D)RL engine outputs a changer that is decoded, and executed by the agent; a reward on the changer is computed; (b) **exploitation**, learned Q-table or Deep Q-network is used by the agent to decide the changer to execute given the current state.

reached. Traces in this setting are online (as in the RV case) in the sense that the State-Changer Sequence feeds the learning engine one state at a time; also, the way they are used and memorized depends on the learning engine, which is an opaque service for the agent. The reward machine, by contrast, resides on the symbolic side: it maps environment observations (such as goal achievement, timeout, or intermediate progress) to numeric reward signals, giving the agent declarative control over the learning objective. The changer is passed back to the SUA because the agent must physically execute the selected action in its environment before the next state can be observed, thus closing the perception-action loop. Throughout this paper we use the term "reward machine" to broadly denote a structured, declarative specification of the reward signal. In our setting, the reward machine consists of rules over agent beliefs, rather than a full finite-

state automaton over propositional symbols as in Toro Icarte et al. [56,57]. We adopt this term because the rules are evaluated in the symbolic layer and produce the scalar reward through symbolic inference. This symbolic flavor of the reward logic may help explainability, since the criteria for success and failure are transparent and inspectable.

The output of the exploration stage is either a Q-table or a Deep Q-network or any other sub-symbolic representation of the learned transition function, that now the agent can use (exploitation, Figure 4(b)) to ask for the next changer to execute, given the current state.

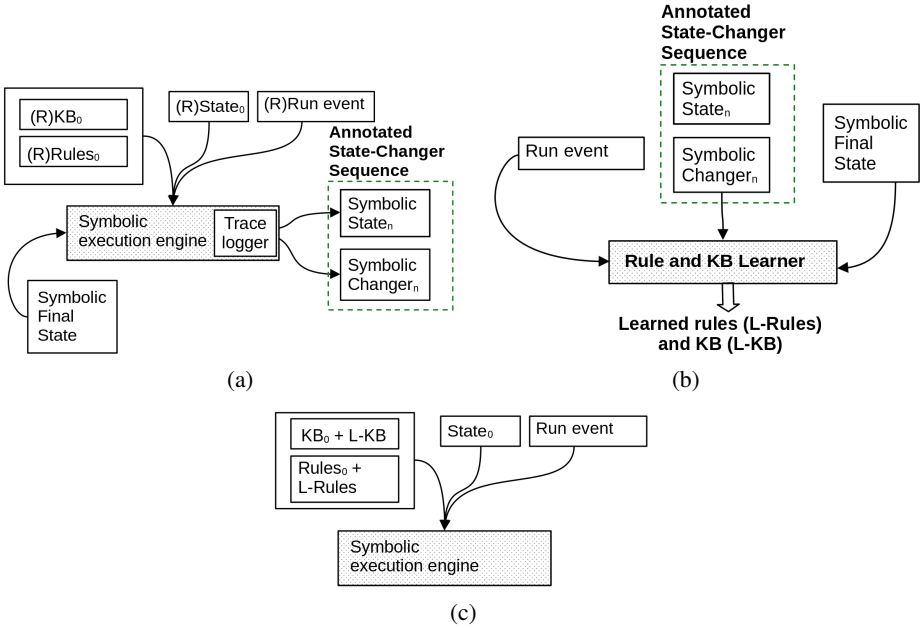

Fig. 5. Learning rules from traces: (a) **exploration**, the agent undergoes some random perturbation of its components, and traces are annotated with whether they represent a success or failure trace w.r.t. reaching a final state; (b) **learning**, annotated traces feed a learner: starting from randomly generated rules, rules and KBs leading to the final state are learned; (d) **exploitation**, learned rules and KBs enter the agent code; in this stage traces may or may not be logged, depending on whether the learning process is iterative or not.

Learning rules from traces. Figure 5 shows a more original way to conceive learning in a symbolic setting.

Our idea is to mimic the exploration-exploitation paradigm by randomly changing the agent's rules, or KB, or run event, or all of them, and 'see what happens'. This approach is also inspired by genetic algorithms, where random mutations take place and individuals that are generated by these mutation, and that are more fitting the survival needs, are allowed to reproduce. Let us suppose that the KB of an agent programmed in an AgentSpeak(L)-style simplified language, consists of

```
my_position(P).
```

where `P` might be kitchen, office, garden and

```
holding(H).
```

where `H` might be cup, phone, etc.
The only rule available for the agent is

```
drink_coffee :  true <-
   move_to_kitchen; find_cup; pour_coffee_in_cup; drink.
```

Now, let us suppose that by randomly generating new rules with available, syntactically correct building blocks (conditions, changers)

```
drink_coffee :  my_position(kitchen) <-
   find_cup; pour_coffee_in_cup; drink.
```

is generated (Figure 5(a)). This rule allows the agent to drink coffee and to avoid to call the 'move to the kitchen' changer, if it is already there. If the execution trace of the agent keeps track of initial rules, it will show that this new rule leads – in an efficient way – to a final state where coffee can be drunk. Having many traces coming from many different runs and a suitable rule learning strategy (Figure 5(b)) will lead to learn robust and reusable rules and KBs, that may then be added to the agent's code, for exploitation (Figure 5(c)).

While an efficient implementation of the rule learning engine might be based on machine learning approaches, other approaches may be adopted if explainability, safety and transparency are hard requirements. Inductive Logic Programming [26,66] is one of them, but also explainable process mining [59] may be used. While both approaches might struggle in scaling when the amount of traces is huge, we look at them with interest.

4 Towards a NOVAE Instance: NOVAE VEsNAe

Maintaining verifiability and explainability while integrating neural learning is a key challenge that NOVAE aims to tackle. Hybrid neuro-symbolic approaches address this challenge by using symbolic knowledge to constrain learning through logical preconditions or learned symbolic representations. Execution traces serve multiple purposes: providing training data for neural components, evidence for symbolic verification, and foundations for explaining learned behaviors. This diversity of trace-based learning motivates the unified treatment in Section 3. In this section, we provide examples of the four key components of the NOVAE vision, namely the trace logger, the RV monitor, the explainer, and the mechanism for learning. Although we are still far from their full integration on top of VEsNA, these examples suggest that the implementation of NOVAE VEsNAe is feasible.

VEsNA [39,48] is a general-purpose agent-based framework for managing virtual environments through natural-language. It integrates a natural language interface for agent to agent and agent to human interaction (ChatBDI [49,50]), a dynamic virtual

environment supporting agents' perception and action (Godot [51] or Unity [85]), and a cognitive framework that enables symbolic reasoning on agents' goals, beliefs, and plans (Jason or JaCaMo [15]).

While the Jason `Agent` class contains all the methods the agent uses to actually make decisions, the `AgArch` class acts as the reasoning orchestrator. VEsNA extends the default `Agent` class connecting it natively to the body situated in the Virtual Reality (VR) environment and managing all the messages between Jason mind and VR body. In addition to these extensions, VEsNA provides a set of features like situated artifacts and communication [47,38] and plans annotated with propensities [46].

4.1 NOVAE VEsNAe Trace Logger

In order to collect information that may be relevant for traces, we extended the Jason `Agent` class and the `AgArch` class in the `vesna` package with logging capabilities.

The extended classes, `LoggedAgent` and `LoggedAgArch` respectively, log perceptions, events, options, intentions and all the selection choices. Together, they implement the *Trace Logger* component foreseen by the NOVAE architecture.

The implementation of a cleaning robot `r1` that looks for garbage in a 2D grid and carries it to `r2` to be burnt is a well-known example of Jason code and – for the sake of clarity of the logged trace – we implemented a further simplified version of it[5].

The log of our simplified version of `r1` at reasoning cycle 3 is shown below. It includes `r1`'s initial beliefs (lines 1 to 5) and plans (lines 6 to 10); perceptions (line 11, when `r1` perceives its change of position and the presence of garbage); and actions for progressing in the execution, such as selecting one event to be processed (line 12).

```
1  2026−02−02−12:05:25 LoggedAgArch reasoningCycleStarting bb {{
2      pos/3=pos(r1,0,0)[source(percept)]← 526334379,
3      pos(r2,1,1)[source(percept)]← 526333743,
4      at/1=at(P)[source(self)] ← (pos(P,X,Y) & pos(r1,X,Y)):1103465977,
5  }}
6  2026−02−02−12:05:25 LoggedAgArch reasoningCycleStarting pl [
7  @p__1[source(self),url("file:r1.asl")] +!check(slots) : not (garbage(r1)) <− ...,
8  @p__2[source(self),url("file:r1.asl")] +!check(slots) : garbage(r1) <− ...,
9  ...
10 ]
11 2026−02−02−12:05:25 LoggedAgArch perceive perceptions [pos(r1,1,0), pos(r2,1,1), garbage(r1)]
12 2026−02−02−12:05:25 LoggedAgent selectEvent events [+pos(r1,1,0)[source(percept)], +garbage(r1)[source(percept)]]
13 2026−02−02−12:05:25 LoggedAgent selectEvent selected +pos(r1,1,0)[source(percept)]
14 2026−02−02−12:05:25 LoggedAgent relevantPlans trigger +pos(r1,1,0)[source(percept)]
15 2026−02−02−12:05:25 LoggedAgent relevantPlans event +pos(r1,1,0)[source(percept)]
16 2026−02−02−12:05:25 LoggedAgent selectEvent events [+garbage(r1)[source(percept)]]
17 2026−02−02−12:05:25 LoggedAgent selectEvent selected +garbage(r1)[source(percept)]
18 2026−02−02−12:05:25 LoggedAgent relevantPlans trigger +garbage(r1)[source(percept)]
19 2026−02−02−12:05:25 LoggedAgent relevantPlans event +garbage(r1)[source(percept)]
20 2026−02−02−12:05:25 LoggedAgArch reasoningCycleFinished
```

Each line of the log contains the timestamp – which is always the same in the above fragment because it refers to the same reasoning cycle –, the class and the method

[5] The Jason code may be found in the Jason repository on GitHub `https://github.com/jason-lang/jason`, in folder `examples/cleaning-robots` while our simplified version is available at `https://github.com/AngeloFerrando/BDIExplainer/tree/main/jason`.

that generated the log and the related piece of information. Depending on the task one may filter out irrelevant information from this exhaustive log, or may log additional data by modifying the `LoggedAgent` and `LoggedAgArch` classes. If this log were used for explainability purposes, beliefs and plans would play the role of *state*, while `+pos(r1,1,0)` would play the role of *changer*. For RV purposes, only `+pos(r1,1,0)`, namely the *changer* that is going to trigger the next plan to execute, would be relevant. In a learning setting, the *state* would be an encoding of the agent's beliefs, and the *changer* would be the action that the DRL component selects, and that Jason must execute.

4.2 NOVAE VEsNAe Verification

NOVAE VEsNAe RV is obtained by instantiating the NOVAE verification view of Section 3.2 on top of the trace-logging infrastructure described in Section 4.1. Verifying at runtime the behavior of Jason-based MASs is something we have been doing since 2012 [4]. We limited RV to communication acts in the beginning, and we extended the property specification language, the RV framework, the application domain over time. We just need to reuse our experience and tools such as the Runtime Monitoring Language, RML [7], to operate on traces collected as discussed in Section 4.1.

For RV, we define a projection π_{RV} that extracts a sequence of changers $\sigma = ch_0 ch_1 \ldots$ from the trace produced by the trace logger discussed in the previous section. Concretely, π_{RV} can retain (depending on the property) for instance: (i) perception changes (e.g., `+garbage(r1)`), (ii) messages sent/received (as in our initial works), (iii) environment actions issued (e.g., `move`, `clean`, `pickUp`), and optionally (iv) key internal steps such as goal adoption/failure or intention selection. This yields a standard RV input trace, while keeping the logger unchanged.

Once π_{RV} is fixed, verification can be performed: *offline*, by parsing the logged trace and replaying σ into a monitor; or *online*, by streaming each newly produced ch_i to the monitor as soon as it is logged. The verification core is the same in both cases: monitors consume the changer stream incrementally and emit verdicts (satisfaction/violation/inconclusive), as in standard RV architectures.

For the purpose above, any of the property languages discussed in Section 2 can be used. In particular, our previous MAS-focused RV work based on parametric trace expressions and related monitoring techniques [7,5,4] naturally fits VEsNA because the extracted events are structured (they carry parameters such as agent id, region, object id) and can be monitored with parametric specifications.

4.3 NOVAE VEsNAe Explainability

In the companion paper 'AgentSpeaX: Explain, Actually' [92] freely available in the EMAS 2026 web site[6] we introduce AgentSpeaX, a conceptual framework for execution-level explainability in AgentSpeak(L) agents, and its instantiation to Jason.

AgentSpeaX defines a formal explanation model grounded in instantiated plan executions, capturing how goals, plans, actions, and belief updates produce observed behavior. By deriving explanations directly from the AgentSpeak(L) execution process,

[6] `https://emas-workshop.github.io/2026/`, accessed on July 9, 2026.

AgentSpeaX provides faithful explanations even under recursion, repeated execution, and failure handling.

The Jason instantiation of AgentSpeaX is based on time-stamped traces of changers that feed a Prolog-based explainer. At the time of writing, the Jason code must be manually translated into the Prolog representation of plans, beliefs, and goals needed by the explainer. Under the assumptions described in [92], this translation can be automated. Traces produced by Jason agents as shown in Section 4.1 may be parsed and cleaned in order to transform relevant elements (timestamps and changers) into the corresponding Prolog representation. While we manually translated real traces from the already mentioned cleaning robot example into Prolog by hand, to quickly come up with a working AgentSpeaX proof of concept, the trace parsing and conversion to Prolog can be developed easily. The trace is an essential input to the explanation generation process. Firstly, the actions performed by the agent are needed to generate explanations since explanations are of actions, and can include those actions' preconditions. Secondly, explanations for why a given plan was selected may need to refer to the agent's beliefs at the point in time where the selection was made.

4.4 NOVAE VEsNAe Learning

To realise the '(Deep) Reinforcement Learning as a Service' vision described in Section 3.4, we extended the VEsNA framework with reinforcement learning (RL) capabilities. The resulting architecture, depicted in Figure 4, cleanly separates two concerns: a *symbolic layer* (implemented in Jason) that encodes the agent's state, computes rewards, and enforces action validity; and an *external DRL service* (implemented in Python) that learns a navigation policy through a Deep Q-Network (DQN). The symbolic layer communicates with the DRL service via a REST API, keeping the two components fully decoupled. To validate this vision, we developed a prototype within the VEsNA framework implementing a navigation scenario where a BDI agent learns to reach goal locations in an office environment comprising interconnected spatial regions with adjacency encoded as Region Connection Calculus (RCC-8) [71] relations in the agent's belief base. RCC is a qualitative spatial formalism that we have previously adopted to represent VEsNA environments [37,45]. The system operates in two phases, illustrated in Figure 4. During *exploration* (Figure 4(a)), the agent observes its current location and goal, encodes them into a numeric state vector, and queries the DRL service for an action. The service selects actions using an ϵ-greedy strategy: with probability ϵ it picks a random valid action (exploration), and otherwise it picks the action with the highest estimated value (exploitation). A symbolically-defined *reward machine* in the Jason layer assigns rewards: a high positive reward for reaching the goal, a small penalty for each intermediate step, and a larger penalty for exceeding a time limit. The DRL service stores each experience and periodically updates its neural network. During *exploitation* (Figure 4(b)), the trained network is loaded and queried for optimal actions without further learning, enabling policy reuse across different goals and improving generalization. Table 3 instantiates the NOVAE trace components for the navigation learning task. An episode terminates when the agent reaches its goal (high positive reward), exceeds the step limit (negative reward), or after each intermediate step (small cost).

SUA	KB	Rules	State	Run Event	Changer
A VEsNA agent learning to navigate	RCC-8 adjacency relations; current position and goal	Plans delegating action selection to DRL service	Goal-conditioned vector $\mathbf{s} = [\mathbf{o}_{\text{curr}} \| \mathbf{o}_{\text{goal}}]$ (one-hot encodings)	Episode start	Movement to an adjacent region, selected by DRL service

Table 3. Trace components for a VEsNA agent learning to navigate.

Symbolic Constraints on Learning. A key design principle is that symbolic knowledge constrains neural exploration. The agent's KB defines valid actions through spatial adjacency relations. Before querying the DRL service, the symbolic layer computes the set of valid actions $\mathcal{A}(s) \subseteq \mathcal{A}$ based on the current region's neighbors. The neural component applies *action masking*: invalid actions receive $-\infty$ Q-values, ensuring they are never selected. This mechanism provides *structural verification* throughout learning [2]: the agent cannot attempt impossible transitions. The symbolic KB acts as a safety shield while the neural component learns optimal behavior within these constraints. Our architecture is designed to bridge learning and explainability through enriched traces. The DRL service returns the selected action along with Q-values for all valid alternatives, enabling to record the selected action and its estimated value, to capture alternative actions with their values for 'Why not?' queries, and to distinguish exploitation from exploration decisions. This design supports post-hoc explanation of learned behavior: given a trace showing the agent chose action a_1 over a_2, an explainer can report that a_1 was selected because its Q-value exceeded that of a_2, indicating higher expected cumulative reward.

Algorithms 1 and 3 in the Appendix formalise the training and inference procedures, respectively. Algorithm 2 provides the masked action selection subroutine used by both.

A working prototype of the navigation learning mechanism is available at `https://github.com/ZahraD-a/Vesna-RL`. All the details of its design, implementation, and evaluation are presented in [28].

We are exploiting neuro-symbolic learning mechanisms in VEsNA also to learn how to adapt mutable personality traits to different situations, depending on past experiences [27,29]. The learning mechanism adopted in these two recent works is Counterfactual Regret Minimization [97,43,19] which, by relying on traces, definitely fits the wide NOVAE VEsNAe vision.

5 Conclusions and Future Work

In this paper we presented the NOVAE vision, characterized by three aspects:

1. **Trace Duality**: A single execution produces one trace serving different purposes: in the learning setting, state-action-reward tuples feed neural learning, while Q-value annotations support symbolic explanation; at the same time, traces – along with snapshots of KB and plans at any given moment – may serve as memories to explain what agents actually did, and observed changers may be monitored at runtime, by checking the trace against the specification of the expected behavior.

2. **Neuro-Symbolic Synergy**: SUAs of interest are characterized by symbolic KB and rules, but learning takes place at a sub-symbolic level. The symbolic KB constrains

exploration through action masking, while the reward machine ensures alignment between learned behavior and declarative goals.

3. **Separation of Concerns**: The external DRL service encapsulates learning logic, enabling policy sharing across agents and separation between reasoning and learning; similarly, explainers and RV monitors are all external modules providing services on demand, decoupling the agents functioning from its monitoring and explanation.

NOVAE VEsNAe, albeit consisting in proofs of concept that are not yet integrated, suggests that the architecture in Figure 4 – coherently supporting logging, verification, explainability, and learning – is practically realizable, with traces providing the unifying substrate for learning and explanation. The NOVAE VEsNAe trace logger produces such unifying substrate, retaining all the information relevant to the agents' functioning.

The future development of NOVAE VEsNAe is to move from the collection of separate proof of concepts, to a unique, coherent working framework. This will allow us to empirically validate our vision, and further shape it.

References

1. Van der Aalst, W.M.P.: Process Mining: Data Science in Action. Springer, 2nd edn. (2016). https://doi.org/10.1007/978-3-662-49851-4
2. Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., Topcu, U.: Safe reinforcement learning via shielding. In: Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence. pp. 2669–2678. AAAI Press, United States (2018). https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17211
3. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 4–16 (2002). https://doi.org/10.1145/503272.503275
4. Ancona, D., Drossopoulou, S., Mascardi, V.: Automatic generation of self-monitoring MASs from multiparty global session types in Jason. In: DALT. Lecture Notes in Computer Science, vol. 7784, pp. 76–95. Springer (2012)
5. Ancona, D., Ferrando, A., Mascardi, V.: Parametric runtime verification of multiagent systems. In: AAMAS. pp. 1457–1459. ACM (2017)
6. Ancona, D., Ferrando, A., Mascardi, V.: Mind the gap! runtime verification of partially observable MASs with probabilistic trace expressions. In: Baumeister, D., Rothe, J. (eds.) Multi-Agent Systems - 19th European Conference, EUMAS 2022, Düsseldorf, Germany, September 14-16, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13442, pp. 22–40. Springer (2022). https://doi.org/10.1007/978-3-031-20614-6_2
7. Ancona, D., Franceschini, L., Ferrando, A., Mascardi, V.: RML: theory and practice of a domain specific language for runtime verification. Sci. Comput. Program. **205**, 102610 (2021)
8. Anjomshoae, S., Najjar, A., Calvaresi, D., Främling, K.: Explainable agents and robots: Results from a systematic literature review. In: AAMAS. pp. 1078–1088 (2019). http://dl.acm.org/citation.cfm?id=3331806
9. Badica, A., Badica, C., Ivanovic, M., Mitrovic, D.: An approach of temporal difference learning using agent-oriented programming. In: 20th International Conference on Control Systems and Computer Science, CSCS 2015, Bucharest, Romania, May 27-29, 2015. pp. 735–742. IEEE (2015). https://doi.org/10.1109/CSCS.2015.71
10. Bakar, N.A., Selamat, A.: Runtime verification of multi-agent systems interaction quality. In: Selamat, A., Nguyen, N.T., Haron, H. (eds.) Intelligent Information and Database Systems - 5th Asian Conference, ACIIDS 2013, Kuala Lumpur, Malaysia, March 18-20, 2013,

Proceedings, Part I. Lecture Notes in Computer Science, vol. 7802, pp. 435–444. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36546-1_45

11. Baldoni, M., Baroglio, C., Micalizio, R., Tedeschi, S.: Accountability in multi-agent organizations: from conceptual design to agent programming. Auton. Agents Multi Agent Syst. **37**(1), 7 (2023). https://doi.org/10.1007/s10458-022-09590-6
12. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Lectures on Runtime Verification - Introductory and Advanced Topics, Lecture Notes in Computer Science, vol. 10457, pp. 1–33. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_1
13. Bemthuis, R.H., Koot, M., Mes, M.R.K., Bukhsh, F.A., Iacob, M.E., Meratnia, N.: An agent-based process mining architecture for emergent behavior analysis. In: 2019 IEEE 23rd International Enterprise Distributed Object Computing Workshop (EDOCW). pp. 54–64 (2019). https://doi.org/10.1109/EDOCW.2019.00022
14. Bento, A., Correia, J., Filipe, R., Araújo, F., Cardoso, J.: Automated analysis of distributed tracing: Challenges and research directions. J. Grid Comput. **19**(1), 9 (2021). https://doi.org/10.1007/S10723-021-09551-5
15. Boissier, O., Bordini, R.H., Hübner, J.F., Ricci, A., Santi, A.: Multi-agent oriented programming with JaCaMo. Science of Computer Programming **78**(6), 747–761 (Jun 2013). https://doi.org/10.1016/j.scico.2011.10.004
16. Bordini, R., Hübner, J., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak Using Jason, vol. 8. John Wiley & Sons, Ltd, United Kingdom (10 2007). https://doi.org/10.1002/9780470061848
17. Bosello, M., Ricci, A.: From programming agents to educating agents - a jason-based framework for integrating learning in the development of cognitive agents. In: Engineering Multi-Agent Systems - 7th International Workshop, EMAS 2019, Montreal, QC, Canada, May 13-14, 2019, Revised Selected Papers. Lecture Notes in Computer Science, vol. 12058, pp. 175–194. Springer (2019). https://doi.org/10.1007/978-3-030-51417-4_9
18. Briola, D., Mascardi, V., Ancona, D.: Distributed runtime verification of JADE and jason multiagent systems with prolog. In: CILC. CEUR Workshop Proceedings, vol. 1195, pp. 319–323. CEUR-WS.org (2014)
19. Brown, N., Sandholm, T.: Solving imperfect-information games via discounted regret minimization. In: The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019. pp. 1829–1836. AAAI Press (2019). https://doi.org/10.1609/AAAI.V33I01.33011829
20. 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). https://doi.org/10.3233/IA-190036
21. Chen, B., Jiang, Z.M.J.: A survey of software log instrumentation. ACM Comput. Surv. **54**(4), 90:1–90:34 (2022)
22. Chen, L., Lu, K., Rajeswaran, A., Lee, K., Grover, A., Laskin, M., Abbeel, P., Srinivas, A., Mordatch, I.: Decision transformer: Reinforcement learning via sequence modeling. In: Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual. pp. 15084–15097 (2021). https://proceedings.neurips.cc/paper/2021/hash/7f489f642a0ddb10272b5c31057f0663-Abstract.html
23. Ciccone, L., Ferrando, A., Ancona, D., Mascardi, V.: Timed trace expressions. In: CILC. CEUR Workshop Proceedings, vol. 2396, pp. 229–241. CEUR-WS.org (2019)

24. Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., Koschke, R.: A systematic survey of program comprehension through dynamic analysis. IEEE Trans. Software Eng. **35**(5), 684–702 (2009). https://doi.org/10.1109/TSE.2009.28
25. Cranefield, S., Oren, N., Vasconcelos, W.W.: Accountability for practical reasoning agents. In: Lujak, M. (ed.) Agreement Technologies (AT). LNCS, vol. 11327, pp. 33–48. Springer (2018). https://doi.org/10.1007/978-3-030-17294-7_3
26. Cropper, A., Dumancic, S.: Inductive logic programming at 30: A new introduction. J. Artif. Intell. Res. **74**, 765–850 (2022). https://doi.org/10.1613/JAIR.1.13507
27. Daoui, Z., Ferrando, A., Gatti, A., Mascardi, V.: Learning social personalities: Counterfactual regret in Pro-AgentSpeak(L) agents. In: Accepted for inclusion in the Proceedings of EUMAS 2026, the 23rd European Conference on Multi-Agent Systems. Springer (2026)
28. Daoui, Z., Gatti, A., Ferrando, A., Mascardi, V.: RAVEN: Extending VEsNA with region-agnostic reinforcement learning for interpretable BDI navigation. In: Proceedings of WOA 2026, the 27th Workshop From Objects to Agents. CEUR (2026)
29. Daoui, Z., Gatti, A., Mascardi, V., Ferrando, A.: One agent, a thousand faces: Learning personality masks for BDI robots. In: Accepted for inclusion in the Proceedings of AREA 2026, the 6th Workshop onAgents and Robots for reliable Engineered Autonomy (2026)
30. Debbi, H.: Counterexamples in model checking – A survey. Informatica (Slovenia) **42**(2), 145–166 (2018)
31. Engelmann, D.C., Ferrando, A., Panisson, A.R., Ancona, D., Bordini, R.H., Mascardi, V.: RV4JaCa - runtime verification for multi-agent systems. In: Cardoso, R.C., Ferrando, A., Papacchini, F., Askarpour, M., Dennis, L.A. (eds.) Proceedings of the Second Workshop on Agents and Robots for reliable Engineered Autonomy, AREA@IJCAI-ECAI 2022, Vienna, Austria, 24th July 2022. EPTCS, vol. 362, pp. 23–36 (2022). https://doi.org/10.4204/EPTCS.362.5
32. Engelmann, D.C., Ferrando, A., Panisson, A.R., Ancona, D., Bordini, R.H., Mascardi, V.: RV4JaCa - towards runtime verification of multi-agent systems and robotic applications. Robotics **12**(2), 49 (2023)
33. Ferrando, A., Ancona, D., Mascardi, V.: Decentralizing MAS monitoring with decamon. In: AAMAS. pp. 239–248. ACM (2017)
34. Ferrando, A., Cardoso, R.C.: Safety shields, an automated failure handling mechanism for BDI agents. In: Faliszewski, P., Mascardi, V., Pelachaud, C., Taylor, M.E. (eds.) 21st International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2022, Auckland, New Zealand, May 9-13, 2022. pp. 1589–1591. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS) (2022). https://doi.org/10.5555/3535850.3536044
35. Ferrando, A., Dennis, L.A., Ancona, D., Fisher, M., Mascardi, V.: Recognising assumption violations in autonomous systems verification. In: AAMAS. pp. 1933–1935. International Foundation for Autonomous Agents and Multiagent Systems Richland, SC, USA / ACM (2018)
36. Ferrando, A., Dennis, L.A., Cardoso, R.C., Fisher, M., Ancona, D., Mascardi, V.: Toward a holistic approach to verification and validation of autonomous cognitive systems. ACM Trans. Softw. Eng. Methodol. **30**(4), 43:1–43:43 (2021)
37. Ferrando, A., Gatti, A., Mascardi, V.: Geometric and spatial reasoning in BDI agents: A survey. In: CILC. CEUR Workshop Proceedings, vol. 3733. CEUR-WS.org (2024)
38. Ferrando, A., Gatti, A., Mascardi, V.: Oops, I Heard That! Situated Communication with Locality-Aware KQML (2025)
39. Ferrando, A., Gatti, A., Mascardi, V.: Integrating Virtual Reality, Chatbots, and BDI Agents: VEsNA Goes Fast!, pp. 289–322. Springer Nature Switzerland, Cham (2026). https://doi.org/10.1007/978-3-032-01082-7_11

40. Ferrando, A., Malvone, V.: Towards the combination of model checking and runtime verification on multi-agent systems. In: Dignum, F., Mathieu, P., Corchado, J.M., de la Prieta, F. (eds.) Advances in Practical Applications of Agents, Multi-Agent Systems, and Complex Systems Simulation. The PAAMS Collection - 20th International Conference, PAAMS 2022, L'Aquila, Italy, July 13-15, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13616, pp. 140–152. Springer (2022). https://doi.org/10.1007/978-3-031-18192-4_12
41. Fikes, R., Nilsson, N.J.: STRIPS: A new approach to the application of theorem proving to problem solving. Artif. Intell. **2**(3/4), 189–208 (1971)
42. Floridi, L., Cowls, J., Beltrametti, M., Chatila, R., Chazerand, P., Dignum, V., Luetge, C., Madelin, R., Pagallo, U., Rossi, F., Schafer, B., Valcke, P., Vayena, E.: AI4People—An Ethical Framework for a Good AI Society: Opportunities, Risks, Principles, and Recommendations. Minds and Machines (Nov 2018). https://doi.org/10.1007/s11023-018-9482-5
43. Foerster, J.N., Farquhar, G., Afouras, T., Nardelli, N., Whiteson, S.: Counterfactual multi-agent policy gradients. In: McIlraith, S.A., Weinberger, K.Q. (eds.) Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018. pp. 2974–2982. AAAI Press (2018). https://doi.org/10.1609/AAAI.V32I1.11794
44. García, J., Fernández, F.: A comprehensive survey on safe reinforcement learning. J. Mach. Learn. Res. **16**, 1437–1480 (2015). https://jmlr.csail.mit.edu/papers/v16/garcia15a.html
45. Gatti, A.: Reason logically, move continuously. In: Ferrando, A., Cardoso, R.C. (eds.) Agents and Robots for reliable Engineered Autonomy. pp. 115–127. Springer Nature Switzerland, Cham (2025)
46. Gatti, A., Casale, F., Mascardi, V., Stucchi, A., Ferrando, A.: VEsNA-Pro: Exploiting BDI agents with propensities for emergent narrative. In: Amato, C., Dennis, L., Mascardi, V., Thangarajah, J. (eds.) Proceedings of the 25th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2025, Paphos, Cyprus, May 25-29, 2026. International Foundation for Autonomous Agents and Multiagent Systems / ACM (2026)
47. Gatti, A., Ferrando, A., Mascardi, V.: Situated agents in action: Extending VEsNA with spatial and grabbable artifacts. In: PAAMS. Lecture Notes in Computer Science, vol. 16031, pp. 348–353. Springer (2025)
48. Gatti, A., Ferrando, A., Mascardi, V.: VEsNA-Toolkit web site (2026). https://github.com/VEsNA-ToolKit, accessed on July 9, 2026
49. Gatti, A., Mascardi, V., Ferrando, A.: ChatBDI: Think BDI, talk LLM. In: AAMAS. pp. 2541–2543. International Foundation for Autonomous Agents and Multiagent Systems / ACM (2025)
50. Gatti, A., Mascardi, V., Ferrando, A.: Let me talk to you! natural language interaction between humans and BDI agents via ChatBDI. In: ECAI. Frontiers in Artificial Intelligence and Applications, vol. 413, pp. 3646 – 3654. IOS Press (2025)
51. Godot foundation: Godot web site (2026). https://godotengine.org/, accessed on July 9, 2026
52. Gunning, D., Vorm, E., Wang, J.Y., Turek, M.: DARPA's explainable AI (XAI) program: A retrospective. Applied AI Letters **2**(4), e61 (2021). https://doi.org/https://doi.org/10.1002/ail2.61
53. He, S., He, P., Chen, Z., Yang, T., Su, Y., Lyu, M.R.: A survey on automated log analysis for reliability engineering. ACM Computing Surveys **54**(6), 130:1–130:37 (2022). https://doi.org/10.1145/3460345
54. High-Level Expert Group on Artificial Intelligence: The assessment list for trustworthy artificial intelligence. https://digital-strategy.ec.europa.eu/en/library/assessment-list-trustworthy-artificial-intelligence-altai-self-assessment (2020)

55. Holland, J.H.: Genetic algorithms. Scientific American **267**(1), 66–73 (1992). http://www.jstor.org/stable/24939139
56. Icarte, R.T., Klassen, T.Q., Valenzano, R.A., McIlraith, S.A.: Using reward machines for high-level task specification and decomposition in reinforcement learning. In: Dy, J.G., Krause, A. (eds.) Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018. pp. 2112–2121. Proceedings of Machine Learning Research, PMLR (2018). http://proceedings.mlr.press/v80/icarte18a.html
57. Icarte, R.T., Klassen, T.Q., Valenzano, R.A., McIlraith, S.A.: Reward machines: Exploiting reward function structure in reinforcement learning. J. Artif. Intell. Res. **73**, 173–208 (2022). https://doi.org/10.1613/JAIR.1.12440
58. IEEE: IEEE Standard for transparency of autonomous systems. IEEE Std 7001-2021 (2022). https://doi.org/10.1109/IEEESTD.2022.9726144
59. Kim, K.P.: XPM: eXplainable Process Mining. In: Nghia, P.T., Thai, V.D., Thuy, N.T., Huynh, V.N., Van Huan, N. (eds.) Advances in Information and Communication Technology. pp. 3–8. Springer Nature Switzerland, Cham (2025)
60. Landauer, M., Onder, S., Skopik, F., Wurzenberger, M.: Deep learning for anomaly detection in log data: A survey. Machine Learning with Applications **12**, 100470 (2023). https://doi.org/10.1016/j.mlwa.2023.100470
61. Langley, P., Meadows, B., Sridharan, M., Choi, D.: Explainable agency for intelligent autonomous systems. In: AAAI Conference on Artificial Intelligence. pp. 4762–4764. AAAI Press (2017). http://aaai.org/ocs/index.php/IAAI/IAAI17/paper/view/15046
62. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebraic Methods Program. **78**(5), 293–303 (2009). https://doi.org/10.1016/j.jlap.2008.08.004
63. Li, B., Peng, X., Xiang, Q., Wang, H., Xie, T., Sun, J., Liu, X.: Enjoy your observability: An industrial survey of microservice tracing and analysis. Empir. Softw. Eng. **27**(1), 25 (2022). https://doi.org/10.1007/S10664-021-10063-9
64. Lim, Y.J., Hong, G., Shin, D., Jee, E., Bae, D.: A runtime verification framework for dynamically adaptive multi-agent systems. In: 2016 International Conference on Big Data and Smart Computing, BigComp 2016, Hong Kong, China, January 18-20, 2016. pp. 509–512. IEEE Computer Society (2016). https://doi.org/10.1109/BIGCOMP.2016.7425981
65. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M.A., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (2015). https://doi.org/10.1038/nature14236
66. Muggleton, S.H., Raedt, L.D.: Inductive logic programming: Theory and methods. J. Log. Program. **19/20**, 629–679 (1994). https://doi.org/10.1016/0743-1066(94)90035-3
67. Nian, Y., Yuan, A., Zhang, H., Li, J., Zhao, Y.: Auditable agents (2026). https://arxiv.org/abs/2604.05485, Accepted at the ACL 2026 Workshop on Towards Knowledgeable Foundation Models (KnowFM)
68. Omicini, A.: Not Just for Humans: Explanation for Agent-to-Agent Communication. In: Vizzari, G., Palmonari, M., Orlandini, A. (eds.) Proceedings of the AIxIA 2020 Discussion Papers Workshop co-located with the the 19th International Conference of the Italian Association for Artificial Intelligence (AIxIA2020), Anywhere, November 27th, 2020. CEUR Workshop Proceedings, vol. 2776, pp. 1–11. CEUR-WS.org (2020). https://ceur-ws.org/Vol-2776/paper-1.pdf
69. Persiani, M., Hellström, T.: The mirror agent model: A bayesian architecture for interpretable agent behavior. In: Calvaresi, D., Najjar, A., Winikoff, M., Främling, K. (eds.) Explainable and Transparent AI and Multi-Agent Systems - 4th International Workshop, EXTRAAMAS 2022, Virtual Event, May 9-10, 2022, Revised Selected Papers. Lecture Notes

in Computer Science, vol. 13283, pp. 111–123. Springer (2022). https://doi.org/10.1007/978-3-031-15565-9_7
70. Pnueli, A.: The temporal logic of programs. In: Proc. 18th Annual Symposium on Foundations of Computer Science (FOCS). pp. 46–57. IEEE Computer Society (1977). https://doi.org/10.1109/SFCS.1977.32
71. Randell, D.A., Cui, Z., Cohn, A.G.: A spatial logic based on regions and connection. In: Nebel, B., Rich, C., Swartout, W.R. (eds.) Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92), Cambridge, MA, USA, October 25-29, 1992. pp. 165–176. Morgan Kaufmann (1992)
72. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Eindhoven, The Netherlands, January 22-25, 1996. Lecture Notes in Computer Science, vol. 1038, pp. 42–55. Springer (1996). https://doi.org/10.1007/BFb0031845
73. Rao, A.S., Georgeff, M.P.: BDI agents: From theory to practice. In: Proceedings of the First International Conference on Multiagent Systems, June 12-14, 1995, San Francisco, California, USA. pp. 312–319. The MIT Press (1995)
74. Reger, G., Havelund, K.: What is a trace? A runtime verification perspective. In: ISoLA (2). Lecture Notes in Computer Science, vol. 9953, pp. 339–355 (2016)
75. Robinette, P., Li, W., Allen, R., Howard, A.M., Wagner, A.R.: Overtrust of robots in emergency evacuation scenarios. In: Human Robot Interaction (HRI). pp. 101–108. IEEE/ACM (2016). https://doi.org/10.1109/HRI.2016.7451740
76. Rodriguez, S., Thangarajah, J., Winikoff, M.: Requirements-based explainability for multi agent systems. In: AAMAS. pp. 2726–2728. International Foundation for Autonomous Agents and Multiagent Systems / ACM (2025)
77. Rodriguez, S., Thangarajah, J., Winikoff, M.: Requirements-based explainability for multi-agent systems. In: AI (1). Lecture Notes in Computer Science, vol. 16370, pp. 246–259. Springer (2025)
78. Roungroongsom, C., Pradubsuwun, D.: Formal verification of multi-agent system based on JADE: A semi-runtime approach. In: Recent Advances in Information and Communication Technology 2015, pp. 297–306. Springer (2015)
79. Rozinat, A., Zickler, S., Veloso, M., van der Aalst, W.M.P., McMillen, C.: Analyzing multi-agent activity logs using process mining techniques. In: Distributed Autonomous Robotic Systems 8, pp. 251–260. Springer (2009). https://doi.org/10.1007/978-3-642-00644-9_22
80. Sánchez, C., Schneider, G., Ahrendt, W., Bartocci, E., Bianculli, D., Colombo, C., Falcone, Y., Francalanza, A., Krstic, S., Lourenço, J.M., Nickovic, D., Pace, G.J., Rufino, J., Signoles, J., Traytel, D., Weiss, A.: A survey of challenges for runtime verification from advanced application domains (beyond software). Formal Methods Syst. Des. **54**(3), 279–335 (2019)
81. Singh, D., Sardiña, S., Padgham, L., James, S.: Integrating learning into a BDI agent for environments with changing dynamics. In: IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011. pp. 2525–2530. IJCAI/AAAI (2011). https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-420
82. Spanoudakis, G., Zisman, A.: Software traceability a roadmap. In: Handbook of Software Engineering and Knowledge Engineering. pp. 395–428 (2005). https://doi.org/10.1142/9789812775245_0014
83. Sterling, L., Shapiro, E.: The Art of Prolog - Advanced Programming Techniques. MIT Press (1986)
84. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, 2nd edn. (2018). http://incompleteideas.net/book/RLbook2020.pdf
85. Unity technologies: Unity web site (2026). https://unity.com/, accessed on July 9, 2026

86. Verhagen, R.S., Neerincx, M.A., Tielman, M.L.: A two-dimensional explanation framework to classify AI as incomprehensible, interpretable, or understandable. In: Explainable and Transparent AI and Multi-Agent Systems (EXTRAAMAS). LNCS, vol. 12688, pp. 119–138. Springer (2021). https://doi.org/10.1007/978-3-030-82017-6_8
87. Weiss, G.M., Hirsh, H.: Learning to predict rare events in event sequences. In: KDD. pp. 359–363. AAAI Press (1998)
88. Winfield, A.F.T., Booth, S., Dennis, L.A., Egawa, T., Hastie, H.F., Jacobs, N., Muttram, R.I., Olszewska, J.I., Rajabiyazdi, F., Theodorou, A., Underwood, M.A., Wortham, R.H., Watson, E.N.: IEEE P7001: A proposed standard on transparency. Frontiers Robotics AI **8**, 665729 (2021). https://doi.org/10.3389/frobt.2021.665729
89. Winikoff, M.: Towards trusting autonomous systems. In: Engineering Multi-Agent Systems (EMAS). LNCS, vol. 10738, pp. 3–20. Springer (2017). https://doi.org/10.1007/978-3-319-91899-0_1
90. Winikoff, M.: Towards engineering explainable autonomous systems. In: Briola, D., Cardoso, R.C., Logan, B. (eds.) Engineering Multi-Agent Systems - 12th International Workshop, EMAS 2024, Auckland, New Zealand, May 6-7, 2024, Revised Selected Papers. Lecture Notes in Computer Science, vol. 15152, pp. 144–155. Springer (2024). https://doi.org/10.1007/978-3-031-71152-7_9
91. Winikoff, M.: Contrastive explanations of BDI agents. In: Amato, C., Dennis, L., Mascardi, V., Thangarajah, J. (eds.) Proceedings of the 25th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2025, Paphos, Cyprus, May 25-29, 2026. International Foundation for Autonomous Agents and Multiagent Systems / ACM (2026)
92. Winikoff, M., Daoui, Z., Gatti, A., Mascardi, V., Ferrando, A.: AgentSpeaX: Explain, actually (2026), Accepted for inclusion in the EMAS 2026 Post Proceedings.
93. Winikoff, M., Sidorenko, G.: Evaluating a mechanism for explaining BDI agent behaviour. In: Agmon, N., An, B., Ricci, A., Yeoh, W. (eds.) Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2023, London, United Kingdom, 29 May 2023 - 2 June 2023. pp. 2283–2285. ACM (2023)
94. Winikoff, M., Sidorenko, G.: Evaluating a mechanism for explaining BDI agent behaviour. In: Calvaresi, D., Najjar, A., Omicini, A., Aydogan, R., Carli, R., Ciatto, G., Mualla, Y., Främling, K. (eds.) Explainable and Transparent AI and Multi-Agent Systems - 5th International Workshop, EXTRAAMAS 2023, London, UK, May 29, 2023, Revised Selected Papers. Lecture Notes in Computer Science, vol. 14127, pp. 18–37. Springer (2023). https://doi.org/10.1007/978-3-031-40878-6_2
95. Winikoff, M., Sidorenko, G., Dignum, V., Dignum, F.: Why bad coffee? explaining BDI agent behaviour with valuings. Artif. Intell. **300**, 103554 (2021)
96. Winikoff, M., Thangarajah, J., Rodriguez, S.: A scoresheet for explainable AI. In: AAMAS. pp. 2171–2180. International Foundation for Autonomous Agents and Multiagent Systems / ACM (2025)
97. Zinkevich, M., Johanson, M., Bowling, M.H., Piccione, C.: Regret minimization in games with incomplete information. In: Platt, J.C., Koller, D., Singer, Y., Roweis, S.T. (eds.) Advances in Neural Information Processing Systems 20, Proceedings of the Twenty-First Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 3-6, 2007. pp. 1729–1736. Curran Associates, Inc. (2007). https://proceedings.neurips.cc/paper/2007/hash/08d98638c6fcd194a4b1e6992063e944-Abstract.html

Appendix

Algorithm 1 Neuro-Symbolic DRL Training (BDI Agent with Reward Machine)

Require: BDI agent with KB predicates `neighbor/2`, `region_id/2`
Require: Episodes N, horizon T, discount γ, batch size B, target update period C
Ensure: Trained Q-network parameters θ

RL Service: Initialize Q_θ, Q_{θ^-}, replay buffer $\mathcal{D}$, $\epsilon \leftarrow 1.0$
for $e = 1$ **to** N **do**
  Jason: $(s_0, g) \leftarrow$ RANDOMDIFFERENTPAIR(V) ▷ Random start/goal
  Jason: TELEPORT(s_0); update belief current_region(s_0)
  for $t = 0$ **to** $T - 1$ **do**
    Jason: $\mathbf{o}_t \leftarrow$ ONEHOT$(s_t) \parallel$ ONEHOT(g) ▷ State encoding
    Jason: $\mathcal{A}_t \leftarrow \{id \mid \text{neighbor}(s_t, r) \text{ and region_id}(r, id)\}$ ▷ Valid actions from KB
    Jason: Compute reward from *previous* transition: ▷ Reward Machine
    if $s_t = g$ **then**
      $r_{t-1} \leftarrow 100,\ d_{t-1} \leftarrow$ **true**
    else if $t = T$ **then**
      $r_{t-1} \leftarrow -10,\ d_{t-1} \leftarrow$ **true**
    else
      $r_{t-1} \leftarrow -1,\ d_{t-1} \leftarrow$ **false**
    end if
    Jason $\xrightarrow{\text{HTTP}}$ **RL Service:** Send $(\mathbf{o}_t, \mathcal{A}_t, r_{t-1}, d_{t-1})$
    RL Service: Store transition $(\mathbf{o}_{t-1}, a_{t-1}, r_{t-1}, \mathbf{o}_t, \mathcal{A}_t, d_{t-1})$ in $\mathcal{D}$
    if $|\mathcal{D}| \geq B$ **then**
      Sample minibatch $\mathcal{B} \sim \mathcal{D}$
      $y_i \leftarrow r_i + \gamma(1 - d_i) \max_{a' \in \mathcal{A}'_i} Q_{\theta^-}(\mathbf{o}'_i)[a']$
      Update θ by minimizing $\sum_i$ HUBER$(Q_\theta(\mathbf{o}_i)[a_i] - y_i)$
    end if
    RL Service: $a_t \leftarrow$ SELECTACTIONMASKED$(Q_\theta, \mathbf{o}_t, \mathcal{A}_t, \epsilon)$ ▷ Algorithm. 2
    RL Service $\xrightarrow{\text{HTTP}}$ **Jason:** Return a_t
    Jason: target $\leftarrow$ IDTOREGION(a_t)
    Jason: VESNA.RUN(target) ▷ Physical movement in Godot
    Godot $\xrightarrow{\text{signal}}$ **Jason:** Perceive s_{t+1} via region_entered
    if d_{t-1} **then**
      break
    end if
  end for
  RL Service: $\epsilon \leftarrow \max(\epsilon_{\min}, \lambda\epsilon)$
  if $e \bmod C = 0$ **then**
    $\theta^- \leftarrow \theta$
  end if
end for
RL Service: SAVECHECKPOINT(θ)

Algorithm 2 SELECTACTIONMASKED

Require: Q-network Q_θ, observation $\mathbf{o}$, valid set $\mathcal{A}$, exploration rate ϵ
Ensure: Action $a \in \mathcal{A}$

```
if RANDOM() < ε then                                  ▷ Exploration
    return UNIFORMSAMPLE(A)
else                                                  ▷ Exploitation with masking
    q ← Q_θ(o)                                        ▷ Q-values for all actions
    m ← [−∞, ..., −∞]                                 ▷ Initialize mask
    m[A] ← 0                                          ▷ Unmask valid actions
    return argmax(q + m)                              ▷ Invalid actions cannot be computed
end if
```

Algorithm 3 DRL-as-a-Service Inference for Embodied BDI Navigation

Require: Pretrained Q_θ, start region s_0, goal region g
Ensure: Trace $\tau = \{\langle s_t, a_t, \mathbf{q}_t|_{\mathcal{A}_t}\rangle\}_{t\geq 0}$

```
Agent starts at region s_0
τ ← [ ]
for t = 0, 1, 2, ... until goal or timeout do
    o_t ← ONEHOT(s_t) || ONEHOT(g)
    A_t ← { id | ∃r : neighbor(s_t, r) and region_id(r, id) }
    Send (o_t, A_t) to RL service
    RL service computes q_t ← Q_θ(o_t) and selects a_t = argmax_{a∈A_t} q_t[a]
    Execute action a_t (walk/teleport to decoded region)
    Observe next perceived region s_{t+1}
    τ.APPEND(⟨s_t, a_t, q_t|_{A_t}⟩)
    if s_{t+1} = g then
        break
    end if
end for
return τ
```

Algorithm 1 describes embodied policy training where the BDI agent physically navigates in the Godot environment while learning. The key distinction is that the *reward machine* resides in the symbolic layer (Jason): the agent computes rewards based on goal achievement and sends them to the external RL service along with the encoded state. The RL service performs ϵ-greedy exploration and DQN updates, while the BDI agent maintains control over what constitutes valid actions (via KB predicates) and success/failure conditions.

Algorithm 2 implements ϵ-greedy action selection with symbolic masking. With probability ϵ, the agent *explores* by uniformly sampling from the valid action set $\mathcal{A}$ (determined by the KB's adjacency predicates). Otherwise, it *exploits* by selecting the action with the highest Q-value among valid alternatives. The masking implemented ensures that invalid actions (those not permitted by the symbolic constraints) are never considered, providing a safety guarantee throughout both training and inference.

Algorithm 3 shows how the trained policy is deployed: the policy is executed greedily while Q-values are logged for explainability purposes.