Autonomous Agents and Multi-Agent Systems (2022) 36:6
https://doi.org/10.1007/s10458-021-09529-3

Formal verification of neural agents in non-deterministic environments

Michael E. Akintunde[1] · Elena Botoeva[1] · Panagiotis Kouvaros[1] · Alessio Lomuscio[1]

Accepted: 14 August 2021

Abstract
We introduce a model for agent-environment systems where the agents are implemented via feed-forward ReLU neural networks and the environment is non-deterministic. We study the verification problem of such systems against CTL properties. We show that verifying these systems against reachability properties is undecidable. We introduce a bounded fragment of CTL, show its usefulness in identifying shallow bugs in the system, and prove that the verification problem against specifications in bounded CTL is in coNExpTime and PSpace-hard. We introduce sequential and parallel algorithms for MILP-based verification of agent-environment systems, present an implementation, and report the experimental results obtained against a variant of the VerticalCAS use-case and the frozen lake scenario.

Keywords Verification · Model checking · Neural agents

1 Introduction

Forthcoming autonomous and robotic systems, including autonomous vehicles, are expected to use machine learning (ML) methods for some of their components. Differently from more conventional AI systems that are programmed directly by engineers, components based on ML are synthesised from data and implemented via neural networks. In an autonomous system these components could execute functions such as perception [38, 48] and control [30, 33]. Employing ML components has considerable attractions in terms of performance (e.g., image classifiers), and, sometimes, ease of realisation (e.g., non-linear controllers). However, it also raises concerns in terms of overall system safety. Indeed, it is known that neural networks, as presently used, are fragile and hard to understand [52].

✉ Alessio Lomuscio
a.lomuscio@imperial.ac.uk

Michael E. Akintunde
michael.akintunde13@imperial.ac.uk

Elena Botoeva
e.botoeva@imperial.ac.uk

Panagiotis Kouvaros
p.kouvaros@imperial.ac.uk

[1] Imperial College London, London, UK

Published online: 09 November 2021

If ML components are to be used in safety-critical systems, including various forthcoming autonomous systems, it is essential that they are verified and validated before deployment; standard practice for conventional software. In some areas of AI, notably multi-agent systems (MAS), considerable research has already addressed the automatic verification of AI systems. These concern the validation of either MAS models [20, 35, 41], or MAS programs [8, 14] against expressive AI-inspired specifications, such as those expressible in epistemic and strategy logic. However, with the exceptions discussed below, there is little work addressing the verification of AI systems synthesised from data and implemented via neural networks. This paper makes a contribution in this direction.

Specifically, we formalise and analyse a closed-loop system composed of a reactive neural agent, synthesised from data and implemented by a feed-forward ReLU-activated neural network (ReLU-FFNN), interacting with a non-deterministic environment. Intuitively, the system follows the usual agent-environment loop of observations (of the environment by the agent) and actions (by the agent onto the environment). To model the complexity and partial observability of rich environments, we assume that the neural agent is interacting with a non-deterministic environment, where non-deterministic updates of the environment's state disallow the agent from fully controlling and fully observing the environment's state. Under these assumptions, differently from all related work, the system's evolution is not linear but branching in the future.

We study the verification problem of these systems against a branching time temporal logic. As is known, scalability is a concern in verification and is also an issue in the case of neural systems. To alleviate these difficulties, we are here concerned with a method that is aimed at finding shallow bugs in the system execution, i.e., malfunctions that are realised within a few steps from the system's initialisation. This kind of analysis has been shown to be of particular importance in applications, see, e.g., bounded model checking (BMC) [12], as, experimentally, bugs are often realised after a limited number of steps. Given this, we focus on a bounded version of CTL, i.e., a language expressing temporal properties realisable in a limited number of execution steps. This allows us to reason about applications where the agents ought to bring about a state of affairs within a finite number of steps, or to verify whether a system remains within safety bounds within a number of steps. This enables us to retain decidability even if we consider infinite domains over the reals for the system's state variables, whereas the verification problem for plain CTL is undecidable, as we show. To further alleviate the difficulty of the verification problem, we also introduce a novel algorithm that checks for the occurrence of bugs in parallel over the execution paths. As we show, in the case of bounded safety specifications, this enables us to return a bug to the user as soon as a violation is identified on any of the branching paths that are explored in parallel. This gives considerable advantages in applications, as we show in an avionics application.

A key feature of the parallel verification procedure that we introduce lies in its completeness: we can determine with precision when a potentially infinite set of states (up to a number of steps from the system's initialisation) satisfies a temporal formula. While this results in a heavier computational cost than some incomplete approaches, there are obvious benefits in precise verification, notably the lack of false positives and false negatives. To the best of our knowledge this is the first sound and complete verification framework for closed-loop neural systems that accounts for non-deterministic, branching temporal evolutions.

The rest of the paper is organised as follows. After discussing related work, in Sect. 4 we formally define systems composed by a neural agent, implemented by a ReLU-FFNN, interacting with non-deterministic environments. We analyse the resulting models built on branching executions and define a bounded version of the branching temporal logic CTL to express specifications of these systems. After defining the verification problem, Sect. 5 introduces monolithic and compositional verification algorithms with a complexity study. In this context we show results ranging from undecidability for unbounded reachability, to coNExpTime upper bound for bounded CTL. We present a toolkit for the practical verification of these systems in Sect. 7, implementing said procedure, providing additional functionalities, and reporting the experimental results obtained. We conclude in Sect. 8.

2 Related work

In [3] a closed-loop neural agent-environment system was put forward and analysed. Like the present contribution the agent was modelled via a ReLU-FFNN. However, differently from here, a simple deterministic environment was considered. As a consequence, the system executions were linear and only bounded reachability properties were analysed. [2] extended this work to neural agents formalised via recurrent ReLU-activated neural networks and verified the resulting linear system executions against bounded LTL properties. In contrast, the model put forward here can account for complex, partially observable environments resulting in branching traces, and the strictly more expressive specification language allows for existential and universal quantification over paths. In addition, while the papers above focus on sequential verification procedures, we here develop a parallel approach specifically tailored at identifying shallow bugs efficiently. This requires novel verification algorithms and mixed-integer linear programming [56] (MILP) encodings.

A number of other proposals have also addressed the issue of closed loop systems. For example, [31] presents an approach based on hybrid systems to analyse a control-plant where neural networks are synthesised controllers. Their approach is incomparable with the one here pursued, since they target sigmoidal activation functions (while we focus on ReLU activation functions). Also their verification procedure is not complete, while completeness is a key objective here. Similarly, [15, 28, 32, 57] present work addressing closed loop systems with learned controllers and focus on reachable set estimation and, hence, incomplete techniques for such systems.

Lastly, there has been recent activity on complete approaches for verifying standalone ReLU-FFNNs [6, 9–11, 17, 26, 27, 34, 36, 37, 40, 44, 53]. The systems considered in these approaches are not closed-loop and do not incorporate the environment. This makes the problems considered there different from those analysed here; for instance no temporal evolution can be considered for neural network-controlled agents interacting with an environment. We refer to [24, 29, 39] for surveys on the emerging area of verification of neural networks.

In comparison with [1], we here report several novel optimisations in the tool with a more efficient encoding of the discussed aircraft collision avoidance scenario. We also evaluate our tool on an additional reinforcement learning scenario.

More broadly, this line of work is related to long standing efforts in bounded model checking [7, 47] that are tailored to finding malfunctions easily accessible from the initial states. While our approach is technically different from BMC, it shares with it the characteristic of being more efficient than full exploration methods when only a fraction of the model needs to be explored.

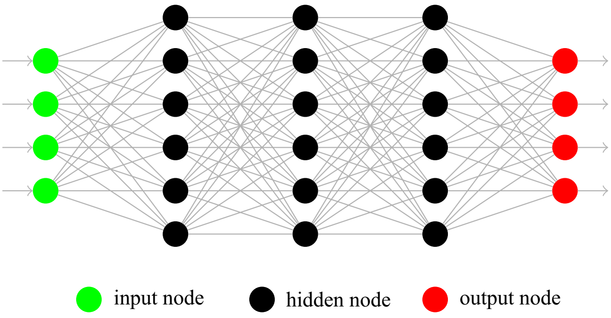


Fig. 1 A feed-forward neural network of 4 input nodes, 3 hidden layers of 6 nodes each and 4 output nodes

3 Background

In this section we summarise basic concepts pertaining to feed-forward ReLU networks and the formalisation of their verification problem in mixed integer linear programming.

3.1 Feed-forward ReLU networks

A *feed-forward neural network (FFNN)* [25] is a vector-valued function $\mathsf{f} : \mathbb{R}^{s_0} \rightarrow \mathbb{R}^{s_L}$ that composes a sequence of $L \geq 1$ *layers*, $\mathsf{f}^1 : \mathbb{R}^{s_0} \rightarrow \mathbb{R}^{s_1}, \ldots, \mathsf{f}^L : \mathbb{R}^{s_{L-1}} \rightarrow \mathbb{R}^{s_L}$. Each layer f^i, $i \in \{1, \ldots L\}$, composes an affine transformation and an *activation* function. Formally, we have for $1 \leq i \leq L$

$$\begin{aligned} \mathsf{f}^i(\mathsf{x}^{i-1}) &\triangleq \mathsf{x}^i, \\ \mathsf{x}^i &\triangleq \mathsf{act}^i(\mathsf{z}^i), \\ \mathsf{z}^i &\triangleq \mathsf{W}^i \mathsf{x}^{i-1} + \mathsf{b}^i, \end{aligned}$$

where:

- x^0 is the input to the network and x^i is the output of the i-th layer.
- z^i is the result of the affine transformation of the i-th layer, known as the *pre-activation* of the layer, for a weight matrix $\mathsf{W}^i \in \mathbb{R}^{s_i \times s_{i-1}}$ and a bias vector $\mathsf{b}^i \in \mathbb{R}^{s_i}$.
- act^i is the activation function of the i-th layer.

Figure 1 gives a graphical representation of a FFNN. Each layer f^i, $i \in \{1, \ldots, L-1\}$, is said to be a *hidden layer*; the last layer f^L of the network is said to be the *output layer*. Each element of each layer f^i is said to be a *node* (see Fig. 2). The weights and biases of the layers are determined during a *training phase* which aims at fitting f to a data set consisting of input-output pairs specifying how the network should behave (see, e.g., [21]) for more details).

Here we are only concerned with FFNNs whose hidden layers use the Rectified Linear Unit (ReLU) and the output layer uses the identity function as activation functions; we abbreviate these networks by ReLU-FFNN. The ReLU activation function is widely used in supervised learning tasks because of its effectiveness in training [43]. The function is defined as

$$\mathrm{ReLU}(z) \triangleq \max(0, z),$$

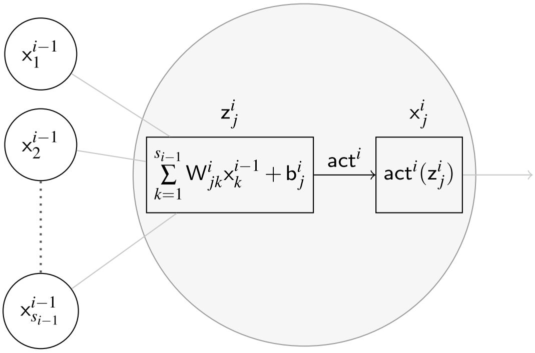


Fig. 2 A node in a feed-forward neural network

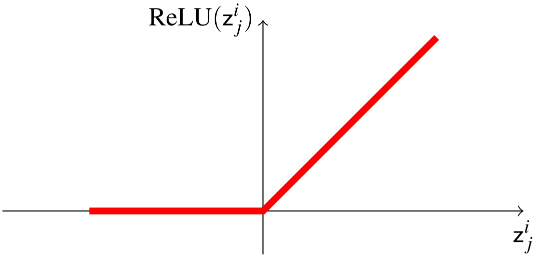


Fig. 3 The ReLU activation function

and is applied element-wise to a pre-activation vector z^i (see Fig. 3). Since the function consists of two linear parts (0, for $z < 0$ and z for $z \geq 0$), it is a *piecewise-linear (PWL)* function; that is, a function whose input domain can be split into a collection of subdomains on each of which it is an affine function. Consequently, since a ReLU-FFNN composes affine transformations with PWL activation functions, ReLU-FFNNs are also PWL functions.

In this paper we are concerned with the *reachability* problem for ReLU-FFNN. The problem is to establish whether there is an admissible input within a possibly uncountable set of inputs for which a given ReLU-FFNN computes an output within a given set of outputs (see e.g., [4, 5, 16, 34])). Formally, we have

Definition 1 (*ReLU-FFNN reachability problem*) Given a ReLU-FFNN $\mathsf{f} : \mathbb{R}^{s_0} \to \mathbb{R}^{s_k}$, a set of inputs $\mathscr{X} \subset \mathbb{R}^{s_0}$ and a set of outputs $\mathscr{Y} \subset \mathbb{R}^{s_k}$, the neural network reachability problem is to determine whether

$$\text{there exists } \mathsf{x} \in \mathscr{X} \text{ such that } \mathsf{f}(\mathsf{x}) \in \mathscr{Y}.$$

The neural network reachability problem is known to be NP-complete [34].

3.2 Mixed integer linear programming

A *mixed integer linear programming* (MILP) is an optimisation problem whereby a linear objective function over real- and integer-valued variables is sought to be minimised subject to a set of linear constraints. Formally, we have

$$\begin{aligned} \min_{\mathsf{x},\mathsf{y}} \quad & c^T\mathsf{x} + d^T\mathsf{y} \\ \textit{subject to} \quad & A\mathsf{x} + B\mathsf{y} \geq b \\ & (\mathsf{x},\mathsf{y}) \in \mathbb{R}^n \times \mathbb{Z}^p, \end{aligned}$$

where $c \in \mathbb{R}^n$, $d \in \mathbb{R}^p$, $A \in \mathbb{R}^{m\times n}$, $B \in \mathbb{R}^{m\times p}$ and $b \in \mathbb{R}^m$.

For the purposes of this paper, we are interested in the *MILP feasibility problem*. This is concerned with checking whether a set of MILP constraints is *feasible*, i.e., whether there exists an assignment to the variables that satisfies all constraints. Therefore, we hereafter assume that the objective function is constant (i.e., it does not depend on the variables), and associate a MILP with a set of linear and typing constraints. It is known that the feasibility problem of MILP is NP-complete [46].

A PWL function can be MILP-encoded using the "Big-M" method. For instance, the pairs (z, x), where $x = \text{ReLU}(z)$ and $z \in [l, u]$ can be found as solutions to the following set of MILP constraints that use a binary variable δ, real-valued variables z and x and constants l and u:

$$x \geq 0, \quad x \geq z, \quad x \leq u \cdot \delta, \quad x \leq z - l \cdot (1 - \delta).$$

Here, when $\delta = 1$, the constraints imply that $x = z$ and $z \geq 0$, and when $\delta = 0$, the constraints imply that $x = 0$ and $z \leq 0$. This approach of "switching off" constraints using large enough constants (l and u in this case) is called the "Big-M" method [22]. Equivalently to this formulation, we can compute the same solutions by making use of *indicator* constraints. These have the form $(\delta = v) \Rightarrow c$, for a binary variable δ, binary value $v \in \{0, 1\}$ and a linear constraint c:

$$\begin{aligned} (\delta = 1) \Rightarrow z \geq 0, \quad & (\delta = 0) \Rightarrow z \leq 0, \\ (\delta = 1) \Rightarrow x = z, \quad & (\delta = 0) \Rightarrow x = 0. \end{aligned}$$

The indicator constraints are read as follows: if $\delta = 1$ then $z \geq 0$ and $x = z$ should hold, and if $\delta = 0$ then $z \leq 0$ and $x = 0$ should hold. Indicator constraints are supported by all major commercial MILP solvers and can be seen as syntactic sugar for Big-M constraints, where one does not have to provide the big M constant in advance. In particular, indicator constraints can be used to naturally express disjunctive cases, cf. monolithic encoding in Sect. 5.

Since ReLU-FFNNs are PWL, the ReLU-FFNN reachability problem has an exact MILP representation: a feasible solution of the corresponding MILP can be used to find an input x^0 from the given set of inputs $\mathscr{X}$ so that $\mathsf{f}(\mathsf{x}^0)$ belongs to the given set of outputs $\mathscr{Y}$ [4, 5, 9, 40, 53]. To define the associated MILP, we assume the following: (i) $\mathscr{X}$ and $\mathscr{Y}$ can be respectively expressed as sets of linear constraints over variables of the inputs and outputs of the network; (ii) a lower bound l^i_j and an upper bound u^i_j for each pre-activation z^i_j have been computed (the bounds can be computed from $\mathscr{X}$ via bound propagation methods, see, e.g., [50, 55]).

Definition 2 (*MILP formulation*) The MILP formulation of the ReLU-FFNN reachability problem for a ReLU-FFNN f, an input set $\mathscr{X}$ and an output set $\mathscr{Y}$ is

$$
\begin{aligned}
&\mathsf{x}^0 \in \mathscr{X} && \text{(input)}\\
&\mathsf{z}^i = \mathsf{W}^i \mathsf{x}^{i-1} + \mathsf{b}^i && \text{(pre-activation)}\\
&\mathsf{x}^i_j \geq 0 && \text{(ReLU)}\\
&\mathsf{x}^i_j \geq \mathsf{z}^i_j && \text{(ReLU)}\\
&\mathsf{x}^i_j \leq \mathsf{u}^i_j \cdot \delta^i_j && \text{(ReLU)}\\
&\mathsf{x}^i_j \leq \mathsf{z}^i_j - \mathsf{l}^i_j \cdot (1 - \delta^i_j) && \text{(ReLU)}\\
&\delta^i_j \in \{0, 1\} \qquad 1 \leq i < L,\ 1 \leq j \leq s_i && \text{(ReLU)}\\
&\mathsf{x}^L = \mathsf{W}^L \mathsf{x}^{L-1} + \mathsf{b}^L && \text{(output)}\\
&\mathsf{x}^L \in \mathscr{Y} && \text{(output)}
\end{aligned}
$$

There exists an input $\mathsf{x}^0 \in \mathscr{X}$ such that $\mathsf{f}(\mathsf{x}^0) \in \mathscr{Y}$ iff the MILP above is feasible.

4 Neural agent-environment systems

In this section we introduce systems with a neural agent operating on a non-deterministic environment (NANES). These are an extension to non-deterministic environments of the deterministic neural agent-environment systems put forward in [3].

In contrast to traditional models of agency, where the agent's behaviour is given in an agent-based programming language, a NANES accounts for the recent shift to synthesise the agents' behaviour from data [33]; we consider agent protocol functions implemented via ReLU-FFNNs [25]. Differently from [3], following the dynamism and unpredictability of the environments where autonomous agents are typically deployed [42], a NANES models interactions of an agent with a partially observable environment. In this setting an agent cannot observe the full environment state, and therefore cannot deterministically predict the effect of any of its actions.

We now proceed to give a formal description of NANES components: a neural agent and a non-deterministic environment. The description closely follows the formalism of *interpreted systems*, a mainstream semantics for multi-agent systems [19]. To this end, we fix a set $S \subseteq \mathbb{R}^m$ of environment states and a set $Act \subseteq \mathbb{R}^n$ of actions, for $m, n \in \mathbb{N}$. We assume that the agent is stateless and that its protocol (also known as action policy) has already been synthesised, e.g., via reinforcement learning [51], and is implemented via a ReLU-FFNN or via a PWL combination of them.

Definition 3 (*Neural Agents*) Let S be a set of environment states.

A *neural agent* (or simply an *agent*) Ag acting on an environment is defined as the tuple $Ag = (Act, prot)$, where:

- Act is a set of actions;
- $prot : S \rightarrow Act$ is a *protocol function* that determines the action the agent will perform given the current state of the environment. Specifically, given ReLU-FFNNs $\mathsf{f}_1, \ldots, \mathsf{f}_h$, $h \geq 1$, $prot$ is a PWL combination of the latter.

When $h = 1$, $prot(s)$ can be defined, e.g., as $\mathsf{f}_1(s)$ for $s \in S$.

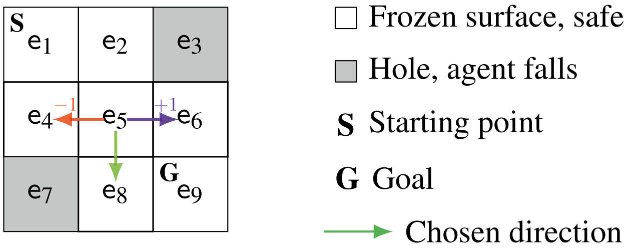


Fig. 4 The FrozenLake surface

The environment is stateful and non-deterministically updates its state in response to the actions of the agent.

Definition 4 (*Non-deterministic Environments*) An *environment* is a tuple $E = (S, t_E)$, where:

- $S \subseteq \mathbb{R}^m$ is a set of states.
- $t_E : S \times Act \rightarrow 2^S$ is a *transition function* that determines the temporal evolution of the state of the environment. Specifically, given the current state of the environment and the current action of the agent, the transition function returns the set of next possible environment states.

Given the above we can now define a closed-loop system comprising of an agent interacting with an environment.

Definition 5 (NANES) A *Neural Agent operating on a Non-Deterministic Environment System (NANES)* is a tuple $\mathscr{S} = (Ag, E, I)$ where $Ag = (Act, prot)$ is a neural agent, $E = (S, t_E)$ is an environment, and $I \subseteq S$ is a closed set[1] of initial states for the environment.

Hereafter we assume the environment's transition function is PWL and its set of initial states is expressible as a set of linear constraints over integer and real-valued variables. Also, to enable its finite MILP representation, we assume that the function's *branching factor* is bounded, i.e., there is a (arbitrarily large) $b \in \mathbb{N}$ such that the cardinality of $t_E(s, a)$ is bounded by b for all $s \in S$ and $a \in Act$.

Example 1 Consider as a running example a variant of the FrozenLake scenario [45], where an agent navigates in a grid world consisting of walkable tiles (frozen surface) and non-walkable ones (holes in the ice) leading to the agent falling into water. The goal of the agent is to reach the goal tile while avoiding the holes. At each step the agent chooses a direction to walk, but since walkable tiles are slippery, the resulting movement direction is uncertain and does not only depend on the agent's choice. Namely, the agent may result moving in any of the three directions: the chosen one and to ones to its left or right, see Fig. 4 for an illustration. We now formalise this scenario as follows.

The agent is defined as $Ag_{\mathsf{fl}} = (Act_{\mathsf{fl}}, prot_{\mathsf{fl}})$, where

[1] By closed set we mean a set containing all of its boundary points.

- $Act_{\mathsf{fl}} = \{\mathsf{d}_1, \mathsf{d}_2, \mathsf{d}_3, \mathsf{d}_4\} \subseteq \mathbb{R}^4$ encodes the directions left (d_1), down (d_2), right (d_3) and up (d_4), where d_i is the vector with 1 at i-th position and 0 everywhere else, and
- $prot_{\mathsf{fl}}$ is the protocol function implemented via a FFNN network.

The non-deterministic environment models the slippery nature of ice. Here we assume a 3×3 grid world, so formalise the environment as $E_{\mathsf{fl}} = (S_{\mathsf{fl}}, t_{E_{\mathsf{fl}}})$, where:

- $S_{\mathsf{fl}} = \{\mathsf{e}_1, \mathsf{e}_2, \dots, \mathsf{e}_9\} \subseteq \mathbb{R}^9$, where e_i is the vector with 1 at i-th position and 0 everywhere else, and
- $t_{E_{\mathsf{fl}}}(s, a) = \{mv(s, a_{-1}), mv(s, a), mv(s, a_{+1})\}$,

where $mv(s, d)$ returns the state s' resulting from moving from state s in direction d, a_{-1} is the direction to the right of a and a_{+1} is the direction to the left of a when looking in the direction a (e.g., $a_{-1} = \mathsf{d}_1$ and $a_{+1} = \mathsf{d}_3$ for $a = \mathsf{d}_2$). To see that $mv(s, d)$ is a PWL function, note that both S and Act are finite sets.

Finally, we define the set of initial states as $I_{\mathsf{fl}} = \{\mathsf{e}_1\}$ and the FrozenLake system as $\mathcal{S}_{\mathsf{fl}} = (Ag_{\mathsf{fl}}, E_{\mathsf{fl}}, I_{\mathsf{fl}})$. □

With each NANES $\mathcal{S}$ we can associate a temporal model $\mathcal{M}_{\mathcal{S}}$ that is used to interpret temporal specifications.

Definition 6 (*Model*) Given a NANES system $\mathcal{S} = (Ag, E, I)$, its associated *temporal model* $\mathcal{M}_{\mathcal{S}}$ is a pair (R, T) where R is the set of environment states *reachable* from I via T, and $T \subseteq R \times R$ is the successor relation defined by $(s, s') \in T$ iff $s' \in t_E(s, prot(s))$.

Example 2 Figure 5 gives a graphical depiction of the temporal model of the FrozenLake system $\mathcal{S}_{\mathsf{fl}}$, assuming that $prot_{\mathsf{fl}}(\mathsf{e}_1) = \mathsf{d}_2$, $prot_{\mathsf{fl}}(\mathsf{e}_2) = \mathsf{d}_1$, $prot_{\mathsf{fl}}(\mathsf{e}_4) = \mathsf{d}_4$, $prot_{\mathsf{fl}}(\mathsf{e}_5) = \mathsf{d}_3$, $prot_{\mathsf{fl}}(\mathsf{e}_6) = \mathsf{d}_2$, $prot_{\mathsf{fl}}(\mathsf{e}_8) = \mathsf{d}_3$, and $prot_{\mathsf{fl}}(\mathsf{e}_9) = \mathsf{d}_2$. □

In the rest of the paper, we assume to have fixed a NANES $\mathcal{S}$ and the associated model $\mathcal{M}_{\mathcal{S}}$. An $\mathcal{M}_{\mathcal{S}}$-*path*, or simply *path*, is an infinite sequence of states $s_1 s_2 \dots$ where $s_i \in R$ and s_{i+1} is a successor of s_i, i.e. $(s_i, s_{i+1}) \in T$, for each $i \geq 1$. Given a path ρ we use $\rho(i)$ to denote the i-th state in ρ. For an environment state $s = (c_1, \dots, c_m)$, we write $\mathsf{paths}(s)$ to denote the set of all paths originating from s and we use $s.d$ to denote its d-th component c_d
.

We verify NANES against properties expressed in a bounded variant of the temporal logic CTL [13]. It is also possible to verify NANES against bounded versions of LTL, but not pursued here. Inspired by Real-Time Computation Tree Logic (RTCTL) [18], formulae of bounded CTL build upon temporal modalities indexed with natural numbers denoting the temporal depth up to which the formula is evaluated.

Definition 7 (*Bounded CTL*) Given a set of environment states $S \subseteq \mathbb{R}^m$, the *bounded CTL specification language over linear inequalities*, denoted $\mathrm{bCTL}_{\mathbb{R}^<}$, is defined by the following BNF:

$$\varphi ::= \alpha \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid EX^k \varphi \mid AX^k \varphi,$$
$$\alpha ::= c_1(d_1) + \cdots + c_l(d_l) \gtrless c,$$

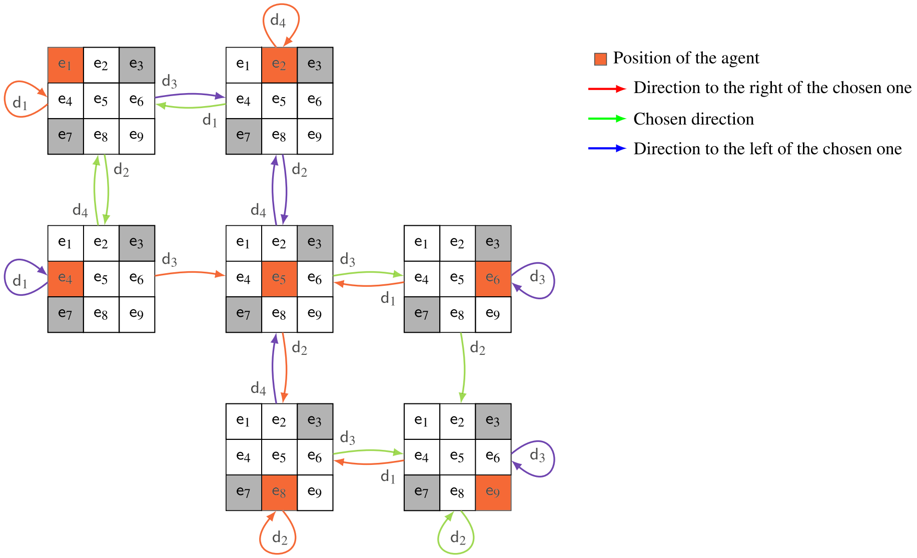

🟧 Position of the agent
Direction to the right of the chosen one
Chosen direction
Direction to the left of the chosen one
e1 e2 e3 e4 e5 e6 e7 e8 e9
d1
d2
d3
d4

Fig. 5 The temporal model of the FrozenLake system $\mathscr{S}_{\mathsf{fl}}$. Each grid depicts the current position of the agent in the environment marked with red colour. Each arrow represents a transition by means of the action labelling the arrow

where $\gtrless \in \{<, >\}$, $d_i \in \{1, \dots, m\}$, $c_i, c \in \mathbb{R}$, and $k \in \mathbb{N}$.

Here atomic propositions α are linear constraints on the components of a state. For instance, the atomic proposition $(d_1) + (d_2) < 2$ states that "the sum of the d_1-st and d_2 -nd components is less than 2." The temporal formula $EX^k\varphi$ stands for "there is a path such that φ holds after k time steps", whereas $AX^k\varphi$ stands for "in all paths φ holds after k time steps". Note that the restriction of bCTL$_{\mathbb{R}^<}$ to strict inequalities is crucial to the verification algorithm introduced in the next section; the algorithm relies on encoding the negation of the specification to check into MILP, which does not support strict inequalities.

Example 3 Consider the FrozenLake scenario from Example 1. We are interested in assessing the safety of the scenario in terms of the following specifications:

$$\varphi^k_{\mathsf{safe}} = AX^1\mathtt{ha} \wedge \cdots \wedge AX^k\mathtt{ha}$$

and

$$\varphi^k_{\mathsf{succ}} = AX^1\mathtt{ha} \wedge \cdots \wedge AX^{k-1}\mathtt{ha} \wedge AX^k\mathtt{goalreached},$$

for various values of k, where $\mathtt{ha} \triangleq \bigwedge_{h\in\{3,7\}}(h) < 0.1$ (i.e., holes voided) and $\mathtt{goalreached} \triangleq (9) > 0.9$. The formula $\varphi^k_{\mathsf{safe}}$ states that in every evolution of the system the agent always avoids a hole within the first k steps. The formula $\varphi^k_{\mathsf{succ}}$ states that in every evolution of the system the agent always avoids a hole within the first $k - 1$ steps and reaches the goal state at the k-th step. Intuitively, the former means that all k-bounded runs are safe (no hole is reached), while the latter means that all k-bounded runs are safe and successful (the goal is reached in the end state). □

We now define the logic $\mathrm{CTL}_{\mathbb{R}^<}$ built from the atoms of $\mathrm{bCTL}_{\mathbb{R}^<}$.

Definition 8 (*CTL*) The *branching-time logic* $\mathrm{CTL}_{\mathbb{R}^<}$ is defined by the following BNF:

$$\varphi{::=}\alpha\big|\neg\varphi \mid \varphi\vee\varphi\big|AX\varphi\big|AF\varphi\big|E(\varphi U\varphi),$$

where α is an atomic proposition in $\mathrm{bCTL}_{\mathbb{R}^<}$.

Comparing $\mathrm{bCTL}_{\mathbb{R}^<}$ to $\mathrm{CTL}_{\mathbb{R}^<}$, we observe that on the one hand $AX^k\varphi$ and $EX^k\varphi$ are expressible, respectively, as $AX(\cdots(AX\varphi)\cdots)$ and $\neg AX(\cdots(AX\neg\varphi)\cdots)$, where AX is applied k times. On the other hand, $\mathrm{CTL}_{\mathbb{R}^<}$ includes the AF ("in all paths eventually") and EU (unbounded until) modalities capable of expressing arbitrary reachability, whereas $\mathrm{bCTL}_{\mathbb{R}^<}$ admits bounded specifications only. Note that, while $\mathrm{bCTL}_{\mathbb{R}^<}$ is clearly less expressive than $\mathrm{CTL}_{\mathbb{R}^<}$, it still captures properties of interest. Notably, *bounded safety* is expressible in $\mathrm{bCTL}_{\mathbb{R}^<}$ as $AG^k safe$ stating that every state on every path is safe within the first k steps.

We interpret $\mathrm{bCTL}_{\mathbb{R}^<}$ formulae on a temporal model as follows.

Definition 9 (*Satisfaction*)

For a model $\mathcal{M}_{\mathcal{S}}$, an environment state s, and a $\mathrm{bCTL}_{\mathbb{R}^<}$ formula φ, the *satisfaction* of φ at s in $\mathcal{M}_{\mathcal{S}}$, denoted $(\mathcal{M}_{\mathcal{S}}, s) \models \varphi$, or simply $s \models \varphi$ when $\mathcal{M}_{\mathcal{S}}$ is clear from the context, is inductively defined as follows:

$$\begin{array}{lll}
s \models c_1(d_1) + \cdots + c_l(d_l) \gtreqless c & \textit{iff} & (\sum_{i=1}^{l} c_i \cdot s.d_i) \gtreqless c; \\
s \models \varphi \vee \psi & \textit{iff} & s \models \varphi \text{ or } s \models \psi; \\
s \models \varphi \wedge \psi & \textit{iff} & s \models \varphi \text{ and } s \models \psi; \\
s \models EX^k\varphi & \text{iff} & \text{there is } \rho \in \mathsf{paths}(s) \text{ such that } \rho(k) \models \varphi; \\
s \models AX^k\varphi & \text{iff} & \text{for all } \rho \in \mathsf{paths}(s) \text{ we have} \rho(k) \models \varphi.
\end{array}$$

We assume the usual definition of satisfaction for $\mathrm{CTL}_{\mathbb{R}^<}$; this can be given as standard by using the atomic case from Definition 9.

Although $\mathrm{bCTL}_{\mathbb{R}^<}$ does not include negation, it still allows us to express arbitrary CTL formulae of bounded temporal depth since it supports all Boolean and temporal operators with their duals. Useful abbreviations of $\mathrm{bCTL}_{\mathbb{R}^<}$ are the temporal modalities $EF^k\varphi$ ("Possibly φ within k steps) and $EG^k\varphi$ ("Possibly φ for k steps):

$$EF^k\varphi \triangleq EX^1\varphi \vee \cdots \vee EX^k\varphi$$
$$EG^k\varphi \triangleq EX^1\varphi \text{ if } k=1; EX^1(\varphi \wedge EG^{k-1}\varphi) \text{ if } k>1.$$

The dual temporal modalities $AG^k\varphi$ and $AF^k\varphi$, prefixed by the universal path quantifier, are analogously defined:

$$AG^k\varphi \triangleq AX^1\varphi \wedge \cdots \wedge AX^k\varphi$$
$$AF^k\varphi \triangleq AX^1\varphi \text{ if } k=1; AX^1(\varphi \vee AF^{k-1}\varphi) \text{ if } k>1.$$

Moreover, bounded until $E(\varphi U^k\psi)$ ("there is a path such that ψ holds within k time steps, and where φ holds up until then") can be defined by the abbreviations

$$E(\varphi U^1\psi) \triangleq \psi \vee (\varphi \wedge EX^1\psi),$$
$$E(\varphi U^k\psi) \triangleq \psi \vee (\varphi \wedge EX^1 E(\varphi U^{k-1}\psi)) \text{ for } k>1,$$

and analogously with $A(\varphi U^k \psi)$.

We also note that the formula $QX^k \varphi$, for $Q \in \{A, E\}$, is equivalent to $QX^1(\cdots(QX^1\varphi)\cdots)$, where QX^1 is applied k times to φ.

A specification φ is said to be *satisfied by* $\mathscr{S}$ if $(\mathscr{M}_{\mathscr{S}}, s) \models \varphi$ for all initial states $s \in I$. We denote this by $\mathscr{S} \models \varphi$. It follows that, for example, to check bounded safety we need to verify that from all (possibly infinitely many) initial states no state (out of possibly infinitely many) within the first k evolutions is an unsafe state. This is the basis of the verification problem that we define below.

Definition 10 (*Verification problem*) Given a NANES $\mathscr{S}$ and a formula φ, determine whether $\mathscr{S} \models \varphi$.

Remark 1 The verification problem is uniquely associated with a *model checking problem* which is to check whether $\mathscr{M}_{\mathscr{S}} \models \varphi$ given $\mathscr{M}_{\mathscr{S}}$ and φ.

However, when the input is specified in terms of a NANES $\mathscr{S}$ and a specification φ, generally, the size (of the relevant part) of the model $\mathscr{M}_{\mathscr{S}}$ grows exponentially in the size of the input.

In the next section we study the decidability and complexity of the verification problem here introduced.

5 The verification problem

In this section we study the verification problem for a NANES against CTL and $\mathrm{bCTL}_{\mathbb{R}^<}$ specifications. First, we show that verifying against CTL formulae is undecidable for deterministic environments and simple reachability properties. In the rest of the section, we focus on bounded CTL, where we develop a decision procedure for the verification problem based on producing a single MILP and checking its feasibility. Then we devise a parallelisable version of the procedure that produces multiple MILPs and that can be particularly efficient at finding counter-examples for bounded safety properties. Following this, we analyse the computational complexity of the verification problem against $\mathrm{bCTL}_{\mathbb{R}^<}$ formulae.

5.1 Unbounded CTL

In this subsection we show undecidability of the verification problem for deterministic NANES against simple reachability properties, where a deterministic NANES is a tuple $(Ag = (Act, prot), E = (S, t_E), I)$, where $|t_E(s, a)| = 1$ for all $s \in S$ and $a \in Act$. The undecidability result for arbitrary NANES and full CTL follows.

Theorem 1 *Verifying deterministic NANES against formulae of the form* $AF\alpha$ *is undecidable.*

Proof We show the result by reduction from the Halting problem which is known to be undecidable. Let $M = \langle Q, \Sigma, \Sigma_0, \delta, q_0, q_a \rangle$ be a Turing machine, where

- $Q = \{1, \ldots, k\}$ is a finite set of states,

- $\Sigma = \{0, 1, 2\}$ is a finite tape alphabet, where we treat 0 as the blank symbol,
- $\Sigma_0 = \{1, 2\}$ is the set of input symbols,
- $q_0, q_a \in Q$ are the initial and accepting states, respectively, and
- $\delta : (Q \setminus \{q_a\}) \times \Sigma \mapsto Q \times \Sigma \times \{\mathrm{L}, \mathrm{R}\}$ is the transition function such that given a state $q \in Q \setminus \{q_a\}$ and a tape symbol σ, $\delta(q, \sigma) = (q', \sigma', m)$ for $q' \in Q$, $\sigma' \in \Sigma$ and $m \in \{\mathrm{L}, \mathrm{R}\}$ with the following meaning: if the Turing machine is in the state q and currently reads symbol σ (i.e., the content of the cell where the head currently is is σ), then write in the current cell σ', move the head left if $m = \mathrm{L}$ and right if $m = \mathrm{R}$, and change the state to q'.

We can assume that the tape of M is open-ended only on the right-hand side, that is, the head never goes to the left of the first cell (cell number 1). The Halting problem defined as "given an input string $\omega_0 \subseteq {\Sigma_0}^h$, decide whether M halts on $\omega_0 = a_1 \dots a_h$, that is, whether M eventually enters q_a" for such Turing machine is known to be undecidable.

We construct a NANES $\mathscr{S} = ((Act, prot), (S, t_E), I)$ and an unbounded temporal formula φ such that $\mathscr{S} \models \varphi$ iff M halts on $\omega_0 = a_1 \dots a_h$.

- each state of $\mathscr{S}$ encodes the current configuration of the Turing machine, that is, the current state, the content of the tape and position of the head on the tape. We account for the position of the head implicitly by storing the content of the tape to the left and to the right of the head. Therefore, the state space S consists of tuples $g = (q, \omega_l, \sigma, \omega_r)$ where

  - $q \in Q$,
  - ω_l represents the left part of the tape and is a real number from [0, 0.3) whose ith digit after the dot stores the content of the ith cell to the left of the head, and hence is one of 0, 1 or 2.
  - $\sigma \in \Sigma$ is the symbol under the head.
  - ω_r represents the right part of the tape and is a real number from [0, 0.3) whose ith digit after the dot stores the content of the ith cell to the right of the head, and hence is one of 0, 1 or 2.

  The left ω_l and right ω_r parts of the tape can be seen as *stacks* with the symbols in the top part being closer to the head, while in the lower part further from the head. In what follows, we refer to them as *left stack* and *right stack*, respectively. Moving the head to the right then corresponds to popping the top symbol from the right stack and pushing σ onto the left stack, and the other way around.
- $I = \{(q_0, 0.0, a_1, 0.a_2 \dots a_h)\}$
- $Act = \{a\}$ for some $a \in \mathbb{R}$.
- the agent's network N computes the constant (hence, linear) function $prot(q, \omega_l, \sigma, \omega_r) = a$.
- the transition function is deterministic and follows closely the transition function δ of M. The main technicality here is to implement δ by suitably updating ω_l, σ and ω_r. Specifically, $t_E((q, \omega_l, \sigma, \omega_r), a) = (q', \omega'_l, \sigma', \omega'_r)$ where

  - $\sigma_l = \lfloor \omega_l \cdot 10 \rfloor$ and $\sigma_r = \lfloor \omega_r \cdot 10 \rfloor$ are the top symbols of ω_l and ω_r, respectively,
  - there exists σ_n and m such that $\delta(q, \sigma) = (q', \sigma_n, m)$,
  - $\omega'_l = (\sigma_n + \omega_l) \cdot 0.1$, $\sigma' = \sigma_r$ and $\omega'_r = \omega_r \cdot 10 - \sigma_r$ if $m = \mathrm{R}$, that is, we push the new symbol σ_n to left stack and pop the top symbol from the right stack, and

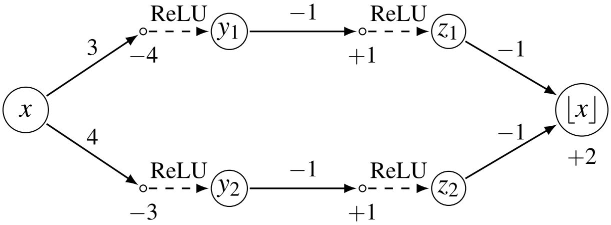
  - $\omega_l' = \omega_l \cdot 10 - \sigma_l$, $\sigma' = \sigma_l$ and $\omega_r' = (\omega_r + \sigma_n) \cdot 0.1$ if $m = \mathrm{L}$, that is, we pop the top symbol from the left stack and push σ_n onto the right stack.

- φ is the reachability specification $EF((1) > |Q| - 1)$, where we assume that all states in Q are numbered from 1 to $|Q|$, and q_a is the state number $|Q|$.

It is straightforward to see that $\mathscr{S}$ and φ are as required.

It can also be seen that t_E is a linearly definable function. We only show an implementation of the function computing the integer part $\lfloor x \rfloor$ of a non-negative real number $x \in [0, 3)$ via a ReLU FFNN $N_{\lfloor \cdot \rfloor}$. Then t_E can be computed by combining it with appropriate linear expressions and conditional statements.

Below we depict the network $N_{\lfloor \cdot \rfloor}$ in which each hidden neuron is split into two nodes, resulting from the linear transformation of the previous layer, and the labelled result of ReLU activation; the weights are drawn on the edges and biases below the nodes. For instance, y_1 is computed from x as $\mathrm{ReLU}(3 \cdot x - 4)$, while $\lfloor x \rfloor$ is computed from z_1 and z_2 as $-z_1 - z_2 + 2$.



To see that this network computes what is intended, observe that the intermediate values y_i and z_i are as follows:

- $y_1 > 0$ iff $x \geq 2$
- $z_1 = 0$ if $x \geq 2$, and $z_1 = 1$ if $x < 2$
- $y_2 > 0$ iff $x \geq 1$
- $z_2 = 0$ if $x \geq 1$, and $z_2 = 1$ if $x < 1$

Conversely, we can construct a NANES $\mathscr{S}' = ((Act', prot'), (S', t_E'), I')$ where t_E' is a linear function and $prot'$ is a PWL function such that $\mathscr{S}' \models \varphi$ iff M halts on ω_0. Namely,

- $Act' = S' = S$, $I' = I$,
- $t_E'(s, a) = a$, and
- $prot'(s) = t_E(s, \_)$.

□

We observe that the above result holds even for strongly restricted NANES where either the protocol or the transition function is linear (but not both at the same time). Intuitively, since every piecewise-linear function can be exactly represented by a ReLU-activated neural network, NANES (even with deterministic environment transition function) are able to simulate recurrent neural networks that are known to be Turing-complete [49].

As a corollary, we obtain undecidability of the verification problem against full CTL.

Corollary 1 *Verifying NANES against* $\mathrm{CTL}_{\mathbb{R}^<}$ *formulae is undecidable.*

5.2 Bounded CTL

We now proceed to investigate the verification problem for the bounded CTL specification language. We start by showing an auxiliary result that allows us to assume without loss of generality that the cardinality of $t_E(s, a)$ is the same for each state s and action a.

Lemma 1 *Given a NANES* $\mathscr{S} = ((Act, prot), (S, t_E), I)$ *and specification* $\varphi \in \mathrm{bCTL}_{\mathbb{R}^<}$, *there is a NANES* $\mathscr{S}' = ((Act, prot'), (S', t'_E), I')$, *such that* $|t'_E(s_1, a_1)| = |t'_E(s_2, a_2)|$ *for all* $s_1, s_2 \in S'$, *and* $a_1, a_2 \in Act$, *and a specification* $\varphi' \in \mathrm{bCTL}_{\mathbb{R}^<}$ *such that* $\mathscr{S} \models \varphi$ *iff* $\mathscr{S}' \models \varphi'$.

Proof Consider $b = \max_{s\in S, a\in Act} |t_E(s, a)|$, following the assumption on boundedness of $|t_E(s, a)|$ for all $s \in S$ and $a \in Act$. First, we define a NANES $\mathscr{S}' = ((Act, prot'), (S', t'_E), I')$ so that $|t'_E(s, a)| = b$ for all $s \in S'$, $a \in Act$:

- $S' = S \times \{0, 1\}$ and $I' = I \times \{1\}$. The added dimension indicates whether a state is valid (1) or not (0).
- The agent's protocol function $prot'$ is defined as $prot'((s,f)) = prot(s)$ for each $s \in S$, $f \in \{0, 1\}$.
- The transition function t'_E is defined as
  - $t'_E((s, 1), a) = t_E(s, a) \times \{1\} \cup \{(s_1, 0), \dots, (s_{b-l}, 0)\}$,
  - $t'_E((s, 0), a) = \{(s_1, 0), \dots, (s_b, 0)\}$

  for $s \in S$, $a \in Act$, $|t_E(s, a)| = l$ and $s_1, \dots, s_b$ pairwise distinct states from S.

Now, suppose that $S = \mathbb{R}^m$. We set specification φ' to be the formula in $\mathrm{bCTL}_{\mathbb{R}^<}$ obtained from φ by replacing each atomic proposition α with $\alpha \wedge ((m + 1) > 0.9)$. It is straightforward to see that $\mathscr{S} \models \varphi$ iff $\mathscr{S}' \models \varphi'$, and hence, φ' is as required. □

In the rest of this section we assume that $|t_E(s, a)| = b$ for all s and a, and that t_E is given as b piecewise-linear (PWL) functions $t_i : \mathbb{R}^{m+n} \to \mathbb{R}^m$. To see that the latter is possible, note that, assuming an ordering (e.g., lexicographical) on the elements of $\mathbb{R}^m$, we can define t_i to return the i-th element of the result by t_E, which is clearly a PWL function. Also note that this assumption is used when devising the verification procedure presented below.

The procedures that we put forward recast the verification problem to the MILP feasibility problem (see Sect. 3 for preliminaries on MILP). Given a MILP program π, we use $vars(\pi)$ to denote the set of variables in π. Denote by $\mathfrak{a}$ the *assignment function* $\mathfrak{a} : vars(\pi) \to \mathbb{R}$, which defines the specific (binary, integer or real) value assigned to a MILP program variable. We write $\mathfrak{a} \models \pi$ if $\mathfrak{a}$ *satisfies* π, i.e., if $\mathfrak{a}(\delta) \in \{0, 1\}$ for each binary variable δ, $\mathfrak{a}(\iota) \in \mathbb{N}$ for each integer variable ι, and all constraints in π are satisfied. Hereafter, we denote by boldface font tuples $\mathbf{x}$ of MILP variables (of length m for $S \subseteq \mathbb{R}^m$ the set of environment states) representing an environment state and call them *state variables*.

As a stepping stone in our procedures, we encode the computation of a successor environment state as a composition of the protocol function *prot* and of the transition functions t_i. By assumption, *prot* and each t_i is a PWL function, and so the predicate $\mathbf{y} = t_i(\mathbf{x}, prot(\mathbf{x}))$ is expressible as a set of MILP constraints by means of the Big-M

$$\begin{aligned}
\pi_{\mathscr{S},\alpha}(\mathbf{x}) &= \{C_\alpha(\mathbf{x})\}, \text{ where } C_\alpha(\mathbf{x}) \text{ is defined as } c_1 x_{d_1} + \cdots + c_l x_{d_l} \gtrless c \text{ for } \alpha = c_1(d_1) + \cdots + c_l(d_l) \gtrless c \text{ and } \mathbf{x} = (x_1, \ldots, x_m),\\
\pi_{\mathscr{S},\varphi_1 \vee \varphi_2}(\mathbf{x}) &= (\delta = 1) \Rightarrow \pi_{\mathscr{S},\varphi_1}(\mathbf{x}) \cup (\delta = 0) \Rightarrow \pi_{\mathscr{S},\varphi_2}(\mathbf{x}),\\
\pi_{\mathscr{S},\varphi_1 \wedge \varphi_2}(\mathbf{x}) &= \pi_{\mathscr{S},\varphi_1}(\mathbf{x}) \cup \pi_{\mathscr{S},\varphi_2}(\mathbf{x}),\\
\pi_{\mathscr{S},EX\varphi}(\mathbf{x}) &= \textstyle\bigcup_{i=1}^{b} (\delta_i = 1) \Rightarrow C_i(\mathbf{x},\mathbf{y}) \cup \{\delta_1 + \cdots + \delta_b = 1\} \cup \pi_{\mathscr{S},\varphi}(\mathbf{y}),\\
\pi_{\mathscr{S},AX\varphi}(\mathbf{x}) &= C_1(\mathbf{x},\mathbf{y}_1) \cup \cdots \cup C_b(\mathbf{x},\mathbf{y}_b) \cup \pi_{\mathscr{S},\varphi}(\mathbf{y}_1) \cup \cdots \cup \pi_{\mathscr{S},\varphi}(\mathbf{y}_b),
\end{aligned}$$

where the binary variables $\delta, \delta_1, \ldots, \delta_b$, the state variables $\mathbf{y}_1, \ldots, \mathbf{y}_b$, $\mathbf{y}$, and all auxiliary variables in $C_i(\mathbf{x}, \mathbf{y}_i)$ are fresh.

Fig. 6 Monolithic encoding $\pi_{\mathscr{S}\varphi}$ for $\varphi \in \mathrm{bCTL}_{\mathbb{R}\leq}$, where for a set of constraints π, $(\delta = v) \Rightarrow \pi$ abbreviates the set of indicator constraints $\{(\delta = v) \Rightarrow c | c \in \pi\}$

method, which we denote by $C_i(\mathbf{x}, \mathbf{y})$ (note that $\mathbf{x} \cup \mathbf{y} \subset vars(C_i(\mathbf{x}, \mathbf{y}))$). Solutions of $C_i(\mathbf{x}, \mathbf{y})$ represent pairs of consecutive environment states:

Lemma 2 *Let $C_i(\mathbf{x}, \mathbf{y})$ be a MILP program corresponding to $\mathbf{y} = t_i(\mathbf{x}, prot(\mathbf{x}))$. Given two states s and s' in $\mathscr{M}_{\mathscr{S}}$, we have that $s' = t_i(s, prot(s))$ iff there is an assignment $\mathfrak{a}$ to $vars(C_i(\mathbf{x}, \mathbf{y}))$ such that $s = \mathfrak{a}(\mathbf{x})$, $s' = \mathfrak{a}(\mathbf{y})$, and $\mathfrak{a} \vDash C_i(\mathbf{x}, \mathbf{y})$.*

Proof The result follows from the fact that $t_i(\mathbf{x}, prot(\mathbf{x}))$ is a PWL function and that every PWL function can be encoded into a set of MILP constraints. Details can be found in [3]. □

5.2.1 Monolithic encoding

First, we devise a recursive encoding of a NANES and a formula into a single MILP, referred to as *monolithic* encoding, and define the corresponding monolithic verification procedure.

Denote by $\mathrm{bCTL}_{\mathbb{R}\leq}$ the bounded CTL language over atomic propositions α where $\gtrless \in \{\leq, \geq\}$ (i.e., linear constraints over non-strict inequalities). Given a NANES $\mathscr{S}$ and a formula $\varphi \in \mathrm{bCTL}_{\mathbb{R}\leq}$, we construct a MILP program $\pi_{\mathscr{S}\varphi}$, whose feasibility corresponds to the existence of a state in $\mathscr{M}_{\mathscr{S}}$ that satisfies φ. For ease of presentation, and without loss of generality, we assume that φ may contain only the temporal modalities EX^1 and AX^1, for which we write EX and AX, respectively.

We now define the *monolithic* encoding $\pi_{\mathscr{S}\varphi}$.

Definition 11 Given a NANES $\mathscr{S}$ and a formula $\varphi \in \mathrm{bCTL}_{\mathbb{R}\leq}$, their *monolithic MILP encoding* $\pi_{\mathscr{S}\varphi}$ is defined as the MILP program $\pi_{\mathscr{S}\varphi}(\mathbf{x})$, where $\mathbf{x}$ is a tuple of fresh state variables, and $\pi_{\mathscr{S}\varphi}(\mathbf{x})$ is built inductively using the rules in Fig. 6.

The monolithic encoding creates one single MILP that entirely accounts for the semantics of the formula, and in particular, every kind of disjunction (in $\varphi_1 \vee \varphi_2$ and $EX\varphi$) is handled by appropriate MILP constraints. Intuitively, the variables $\mathbf{x}$ in the program $\pi_{\mathscr{S}\varphi}(\mathbf{x})$ refer to the states that satisfy the formula φ. In Fig. 6, the base case $\pi_{\mathscr{S}\alpha}(\mathbf{x})$ for an atom α produces the MILP program consisting of a single linear constraint corresponding to α and using variables in $\mathbf{x}$. Each inductive case depends on the state variables $\mathbf{x}$ but might in turn generate programs for subformulas which depend on freshly created state variables different to $\mathbf{x}$ (such as $\mathbf{y}$, $\mathbf{y}_1$, $\mathbf{y}_2$, etc). All other auxiliary variables employed in the encoding are also fresh, preventing undesirable interactions between unrelated branches of the program.

- Disjunction uses a binary variable δ and two sets of indicator constraints. In a feasible assignment $\mathfrak{a}$, when δ is 1, φ_1 is satisfied and the values of $\mathbf{x}$ are assigned according to φ_1, while when δ is 0, φ_2 holds and the values of $\mathbf{x}$ are assigned as per φ_2.
- We encode conjunction as the union of the constraints for each of the conjuncts, which all must be satisfied at the same time.
- We encode an operator EX by a b-ary disjunction using binary variables $\delta_1, \dots, \delta_b$. Each of the possible b next states is chosen by activating one of δ_i hence ensuring that the relevant $C_i(\mathbf{x}, \mathbf{y})$ is satisfied. The variables $\mathbf{y}$ refer to the successor state which must satisfy φ, therefore, the subprogram for φ depends on $\mathbf{y}$. Notably, only one copy of $\pi_{\mathscr{S},\varphi}$ is required.
- To satisfy $AX\varphi$, all b possible successor states should satisfy φ, and so we take the union of all $C_i(\mathbf{x}, \mathbf{y}_i)$ and of b copies of $\pi_{\mathscr{S},\varphi}$, each depending on one of the successor state variables $\mathbf{y}_i$.

Note that the size of $\pi_{\mathscr{S},\varphi}$ may grow exponentially due to b repetitions of $\pi_{\mathscr{S},\psi}$ in $\pi_{\mathscr{S},AX\psi}(\mathbf{x})$; for $\varphi = AX^k\alpha$, the size of $\pi_{\mathscr{S},\varphi}$ is $O(k \cdot b^k \cdot |\mathscr{S}|)$. The same estimate works in the general case for the temporal bound k of φ. On the other hand, when φ contains no AX operator, the size of $\pi_{\mathscr{S},\varphi}$ remains polynomial $O(k \cdot b \cdot |\mathscr{S}|)$.

We can prove that $\pi_{\mathscr{S},\varphi}$ is as intended.

Lemma 3 *Given a NANES $\mathscr{S}$, a formula $\varphi \in \mathrm{bCTL}_{\mathbb{R}_\leq}$ and a state s in $\mathscr{M}_{\mathscr{S}}$, the following are equivalent:*

1. $s \models \varphi$.
2. *There exists an assignment $\mathfrak{a}$ to $vars(\pi_{\mathscr{S},\varphi}(\mathbf{x}))$ such that $\mathfrak{a} \models \pi_{\mathscr{S},\varphi}(\mathbf{x})$ and $s = \mathfrak{a}(\mathbf{x})$.*

Proof Let s be a state in $\mathscr{M}_{\mathscr{S}}$. We prove the statement by induction on the structure of φ. Clearly, the thesis holds when φ is an atomic proposition.

Suppose that the thesis holds for φ_1 and φ_2. Consider the following cases:

$$\begin{aligned} \varphi =& \varphi_1 \vee \varphi_2. \text{ Then} \\ & s \models \varphi_1 \vee \varphi_2 \Leftrightarrow \\ & s \models \varphi_1 \text{ or } s \models \varphi_2 \Leftrightarrow \end{aligned}$$

there exists $\mathfrak{a}_1$ s.t. $\mathfrak{a}_1 \models \pi_{\mathscr{S},\varphi_1}(\mathbf{x})$, $\mathfrak{a}_1(\delta) = 1$ and $s = \mathfrak{a}_1(\mathbf{x})$, or there exists $\mathfrak{a}_2$ s.t. $\mathfrak{a}_2 \models \pi_{\mathscr{S},\varphi_2}(\mathbf{x})$, $\mathfrak{a}_1(\delta) = 0$ and $s = \mathfrak{a}_2(\mathbf{x}) \Leftrightarrow$

there exists $\mathfrak{a}$ s.t. $\mathfrak{a} \models \pi_{\mathscr{S},\varphi_1 \vee \varphi_2}(\mathbf{x})$, and $s = \mathfrak{a}(\mathbf{x})$.

$$\begin{aligned} \varphi =& \varphi_1 \wedge \varphi_2. \text{ Then} \\ & s \models \varphi_1 \wedge \varphi_2 \Leftrightarrow \\ & s \models \varphi_1 \text{ and } s \models \varphi_2 \Leftrightarrow \end{aligned}$$

there exists $\mathfrak{a}_1$ s.t. $\mathfrak{a}_1 \models \pi_{\mathscr{S},\varphi_1}(\mathbf{x})$ and $s = \mathfrak{a}_1(\mathbf{x})$, and there exists $\mathfrak{a}_2$ s.t. $\mathfrak{a}_2 \models \pi_{\mathscr{S},\varphi_2}(\mathbf{x})$ and $s = \mathfrak{a}_2(\mathbf{x}) \Leftrightarrow^{vars(\pi_{\mathscr{S},\varphi_1}(\mathbf{x})) \cap vars(\pi_{\mathscr{S},\varphi_2}(\mathbf{x})) = \mathbf{x}}$

$$\mathfrak{a}_1 \cup \mathfrak{a}_2 \models \pi_{\mathscr{S},\varphi_1}(\mathbf{x}) \cup \pi_{\mathscr{S},\varphi_2}(\mathbf{x}).$$

$$\varphi = EX\varphi_1. \text{ Then}$$
$$s \models EX\varphi_1 \Leftrightarrow$$

there exists a successor s' of s in $\mathscr{M}_{\mathscr{S}}$ s.t. $s' \models \varphi_1 \Leftrightarrow$

there exists $i \in \{1, \dots, k\}$ and an assignment $\mathfrak{a}_i$ such that $\mathfrak{a}_i(\delta_i) = 1$, $\mathfrak{a}_i(\delta_j) = 0$ for $j \neq i$, $\mathfrak{a}_i \models C_i(\mathbf{x}, \mathbf{y})$, $\mathfrak{a}_i(\mathbf{x}) = s$ and $\mathfrak{a}_i(\mathbf{y}) = s'$, and there exists an assignment $\mathfrak{a}'$ s.t. $\mathfrak{a}' \models \pi_{\mathscr{S},\varphi_1}(\mathbf{y})$ and $\mathfrak{a}'(\mathbf{y}) = s' \Leftrightarrow$

there exists $\mathfrak{a}$ s.t. $\mathfrak{a} \models \pi_{\mathscr{S},EX\varphi_1}(\mathbf{x})$ and $\mathfrak{a}(\mathbf{x}) = s$.

$$\varphi = AX\varphi_1. \text{ Then}$$
$$s \models AX\varphi_1 \Leftrightarrow$$

for every successor s_i of s in $\mathscr{M}_{\mathscr{S}}$ it holds that $s_i \models \varphi_1, i = 1, \dots, k, \Leftrightarrow$

for every successor s_i of s in $\mathscr{M}_{\mathscr{S}}$ it holds that there exists $\mathfrak{a}_i$ s.t. $\mathfrak{a}_i \models \pi_{\mathscr{S},\varphi_1}(\mathbf{y}_i)$ and $\mathfrak{a}_i(\mathbf{y}_i) = s_i, i = 1, \dots, k, \Leftrightarrow$

there exists $\mathfrak{a}$ s.t. $\mathfrak{a} \models \pi_{\mathscr{S},AX\varphi_1}(\mathbf{x})$ and $\mathfrak{a}(\mathbf{x}) = s$. □

Finally, we are ready to devise a procedure that solves the verification problem by checking feasibility of the monolithic MILP encoding for the negation of the property to be verified together with a restriction to the initial states of $\mathscr{S}$, $\pi_{\mathscr{S},\mathsf{NNF}(\neg\varphi)\wedge\varphi_I}$. The idea here is to look for a proof that the property is not satisfied by a state $s \in I$. A feasible solution to $\pi_{\mathscr{S},\mathsf{NNF}(\neg\varphi)\wedge\varphi_I}$ then provides such a proof in the form of a counter-example. Conversely, infeasibility implies that no counter-example could be found, and so the property is satisfied.

The procedure is given by Algorithm 1. Recall that strict inequalities are not supported in the MILP solver. Note that by our assumption the set I of initial states is closed and expressible as a set of linear constraints. Therefore, we can represent I by a Boolean formula from $\text{bCTL}_{\mathbb{R}_\leq}$ (i.e., a formula without temporal operators). For instance, the hyper-rectangle $[l_1, u_1] \times \cdots \times [l_m, u_m]$ is represented by the formula

$$(1) \geq l_1 \wedge (1) \leq u_1 \wedge \cdots \wedge (m) \geq l_m \wedge (m) \leq u_m.$$

Further note that we only pass $\neg\varphi$ (the negation of the specification) to the encoding in *negation normal form* (NNF). In this process, negation is eliminated by pushing it down and through the atoms resulting in all strict inequalities of atoms of the original specification φ being converted to non-strict inequalities. Therefore, $\varphi' = \mathsf{NNF}(\neg\varphi) \wedge \varphi_I$ is a formula from $\text{bCTL}_{\mathbb{R}_\leq}$, so $\pi_{\mathscr{S},\varphi'}$ is well-defined and can be processed by an MILP solver.

Soundness and completeness of the verification procedure relies on Lemma 3 and is shown by the following.

Theorem 2 *Given a NANES $\mathscr{S}$ and a formula $\varphi \in$ $\text{bCTL}_{\mathbb{R}_<}$, Algorithm 1 returns False iff $\mathscr{S} \not\models \varphi$.*

Proof Suppose that Algorithm 1 returns *False*. It follows that $\pi_{\mathscr{S},\neg\varphi\wedge\varphi_I}(\mathbf{x})$ is feasible, and there exists an assignment $\mathfrak{a}$ to $vars(\pi_{\mathscr{S},\neg\varphi\wedge\varphi_I}(\mathbf{x}))$ such that $\mathfrak{a} \models \pi_{\mathscr{S},\neg\varphi\wedge\varphi_I}(\mathbf{x})$. Take $s = \mathfrak{a}(\mathbf{x})$. Since φ_I is a Boolean formula (without temporal operators), it is straightforward to see that $s \in I$. Therefore s is a state in $\mathscr{M}_{\mathscr{S}}$, and by Lemma 3 we have that $s \models \varphi_I$ and $s \models \neg\varphi$. It follows that $s \not\models \varphi$, and consequently, $\mathscr{S} \not\models \varphi$. Conversely, if there exists $s \in I$ such that $s \not\models \varphi$, we obtain that there is an assignment satisfying $\pi_{\mathscr{S},\neg\varphi\wedge\varphi_I}(\mathbf{x})$, and therefore Algorithm 1 returns *False*. □

$$\begin{aligned}
\Pi_{\mathscr{S},\alpha}(\mathbf{x}) &= \{[C_\alpha(\mathbf{x})]\}, \\
\Pi_{\mathscr{S},\varphi_1\vee\varphi_2}(\mathbf{x}) &= \Pi_{\mathscr{S},\varphi_1}(\mathbf{x}) \cup \Pi_{\mathscr{S},\varphi_2}(\mathbf{x}), \\
\Pi_{\mathscr{S},\varphi_1\wedge\varphi_2}(\mathbf{x}) &= \Pi_{\mathscr{S},\varphi_1}(\mathbf{x}) \times \Pi_{\mathscr{S},\varphi_2}(\mathbf{x}), \\
\Pi_{\mathscr{S},EX\varphi}(\mathbf{x}) &= \bigcup_{i=1}^{b} \{[C_i(\mathbf{x},\mathbf{y})]\} \times \Pi_{\mathscr{S},\varphi}(\mathbf{y}), \\
&\qquad \text{where the state variables } \mathbf{y} \text{ and all remaining} \\
&\qquad \text{variables in } C_i(\mathbf{x},\mathbf{y}) \text{ are fresh,} \\
\Pi_{\mathscr{S},AX\varphi}(\mathbf{x}) &= \bigtimes_{i=1}^{b} \{[C_i(\mathbf{x},\mathbf{y}_i)]\} \times \Pi_{\mathscr{S},\varphi}(\mathbf{y}_i), \\
&\qquad \text{where the state variables } \mathbf{y}_1,\ldots,\mathbf{y}_b \text{ and all re-} \\
&\qquad \text{maining variables in } C_i(\mathbf{x},\mathbf{y}_i) \text{ are fresh.}
\end{aligned}$$

Fig. 7 Compositional encoding $\Pi_{\mathscr{S},\varphi}$ for $\varphi \in \mathrm{bCTL}_{\mathbb{R}_\leq}$

Algorithm 1 The monolithic MILP verification procedure.

```
procedure MONO-VERIFY($\mathscr{S}$, $\varphi$)
    Input: NANES $\mathscr{S} = (Ag, E, I)$; formula $\varphi \in \mathrm{bCTL}_{\mathbb{R}<}$
    Output: True/False
    $\varphi_I \leftarrow$ Boolean $\mathrm{bCTL}_{\mathbb{R}\leq}$ formula representing $I$
    $\varphi' \leftarrow \mathsf{NNF}(\neg\varphi) \wedge \varphi_I$
    $\pi_{\mathscr{S},\varphi'} \leftarrow$ MILP associated with $\mathscr{S}$ and $\varphi'$
    $feasible \leftarrow \mathsf{MILP\_SOLVER}(\pi_{\mathscr{S},\varphi'})$
    return $\neg feasible$
```

5.2.2 Compositional encoding

Observe that due to its handling of disjunctions, the previously introduced monolithic encoding $\pi_{\mathscr{S},\varphi}$ might result in excessively large programs whose feasibility is a computationally expensive task. We now propose a different encoding that instead of delegating disjunction to the MILP solver (the $\varphi_1 \vee \varphi_2$ and $EX\varphi$ cases) creates a separate program for each disjunct, whose feasibility results can be combined to solve the verification problem. More specifically, for a formula φ, we define a set $\Pi_{\mathscr{S},\varphi}$ of MILP programs with the property that there exists a state s in $\mathscr{M}_{\mathscr{S}}$ such that $s \models \varphi$ iff at least one of the programs in $\Pi_{\mathscr{S},\varphi}$ is feasible.

Below, given a set C of linear constraints, we write $[C]$ to denote the respective MILP program. Given sets $A = \{[A_1], \ldots, [A_p]\}$ and $B = \{[B_1], \ldots, [B_q]\}$ of MILP programs, we write $A \times B$ to denote the *product* of A and B computed as $\{[A_i \cup B_j] \,|\, i = 1,..,p,\ j = 1,..,q\}$.

Definition 12 Given a NANES $\mathscr{S}$ and a formula $\varphi \in \mathrm{bCTL}_{\mathbb{R}_\leq}$, their *compositional MILP encoding* $\Pi_{\mathscr{S},\varphi}$ is defined as the set of MILP programs $\Pi_{\mathscr{S},\varphi}(\mathbf{x})$, where $\mathbf{x}$ is a tuple of fresh state variables, and $\Pi_{\mathscr{S},\varphi}(\mathbf{x})$ is built inductively using the rules in Fig. 7.

Following the monolithic encoding in Fig. 6, in Fig. 7 $C_\alpha(\mathbf{x})$ is the linear constraint corresponding to the atomic proposition α defined over $\mathbf{x}$. We use the same convention regarding the state and auxiliary variables of subprograms. In $\Pi_{\mathscr{S},\varphi}$ every program π represents one of the encodings of φ.

- For disjunction we take the union of the two sets of encodings.

- Every encoding of $\varphi_1 \wedge \varphi_2$ consists of an encoding of φ_1 and of an encoding of φ_2, therefore we take the product of the two sets.
- Every encoding of $EX\varphi$ is an encoding of φ extended with the constraints $C_i(\mathbf{x}, \mathbf{y})$ for a single i.
- Every encoding of $AX\varphi$ consists of b (possibly different) encodings of φ extended with the constraints $C_i(\mathbf{x}, \mathbf{y}_i)$ for $i = 1, \ldots, b$.

The set $\Pi_{\mathscr{S},\varphi}$ grows exponentially with the temporal depth of φ; however each program in the set can be smaller than the monolithic MILP $\pi_{\mathscr{S},\varphi}$.

Similarly to Lemma 3 we can prove that $\Pi_{\mathscr{S},\varphi}$ is as intended.

Lemma 4 *Given a NANES $\mathscr{S}$, a formula $\varphi \in \mathrm{bCTL}_{\mathbb{R}_\leq}$ and a state s in $\mathscr{M}_{\mathscr{S}}$, the following are equivalent:*

1. $s \vDash \varphi$.
2. *There is a MILP $\pi(\mathbf{x}) \in \Pi_{\mathscr{S},\varphi}(\mathbf{x})$ and an assignment $\mathfrak{a}$ to $vars(\pi(\mathbf{x}))$ such that $s = \mathfrak{a}(\mathbf{x})$ and $\mathfrak{a} \vDash \pi(\mathbf{x})$.*

Proof Let s be a state in $\mathscr{M}_{\mathscr{S}}$. We prove the statement by induction on the structure of φ.

Suppose that the thesis holds for φ_1 and φ_2. Consider the following cases:

$$\varphi = \varphi_1 \vee \varphi_2.\text{ Then}$$
$$s \vDash \varphi_1 \vee \varphi_2 \Leftrightarrow$$
$$s \vDash \varphi_1 \text{ or } s \vDash \varphi_2 \Leftrightarrow$$

there exist $\pi(\mathbf{x}) \in \Pi_{\mathscr{S},\varphi_1}(\mathbf{x})$ and an assignment $\mathfrak{a}_1$ such that $\mathfrak{a}_1 \vDash \pi(\mathbf{x})$ and $s = \mathfrak{a}_1(\mathbf{x})$, or there exist $\pi(\mathbf{x}) \in \Pi_{\mathscr{S},\varphi_2}(\mathbf{x})$ and an assignment $\mathfrak{a}_2$ such that $\mathfrak{a}_2 \vDash \pi(\mathbf{x})$ and $s = \mathfrak{a}_2(\mathbf{x}) \Leftrightarrow$

there exist $\pi(\mathbf{x}) \in \Pi_{\mathscr{S},\varphi_1}(\mathbf{x}) \cup \Pi_{\mathscr{S},\varphi_2}(\mathbf{x})$ and an assignment a such that $\mathfrak{a} \vDash \pi(\mathbf{x})$ and $\mathfrak{a}(\mathbf{x}) = s \Leftrightarrow$

there exist $\pi(\mathbf{x}) \in \Pi_{\mathscr{S},\varphi_1 \vee \varphi_2}(\mathbf{x})$ and an assignment $\mathfrak{a}$ such that $\mathfrak{a} \vDash \pi(\mathbf{x})$ and $\mathfrak{a}(\mathbf{x}) = s$.

$$\varphi = \varphi_1 \wedge \varphi_2.\text{ Then}$$
$$s \vDash \varphi_1 \wedge \varphi_2 \Leftrightarrow$$
$$s \vDash \varphi_1 \text{ and } s \vDash \varphi_2 \Leftrightarrow$$

there exist $\pi_1(\mathbf{x}) \in \Pi_{\mathscr{S},\varphi_1}(\mathbf{x})$ and an assignment $\mathfrak{a}_1$ such that $\mathfrak{a}_1 \vDash \pi_1(\mathbf{x})$ and $s = \mathfrak{a}_1(\mathbf{x})$, and there exist $\pi_2(\mathbf{x}) \in \Pi_{\mathscr{S},\varphi_2}(\mathbf{x})$ and an assignment $\mathfrak{a}_2$ such that $\mathfrak{a}_2 \vDash \pi_2(\mathbf{x})$ and $s = \mathfrak{a}_2(\mathbf{x})$ $\Leftrightarrow^{vars(\pi_1(\mathbf{x})) \cap vars(\pi_2(\mathbf{x})) = \mathbf{x}}$

there exist $\pi_1(\mathbf{x}) \in \Pi_{\mathscr{S},\varphi_1}(\mathbf{x})$ and an assignment $\mathfrak{a}_1$, $\pi_2(\mathbf{x}) \in \Pi_{\mathscr{S},\varphi_2}(\mathbf{x})$ and an assignment $\mathfrak{a}_2$ such that $\mathfrak{a}_1 \cup \mathfrak{a}_2 \vDash \pi_1(\mathbf{x}) \cup \pi_2(\mathbf{x})$ and $s = \mathfrak{a}_1(\mathbf{x}) = \mathfrak{a}_2(\mathbf{x}) \Leftrightarrow$

there exists an assignment $\mathfrak{a}$ and $\pi(\mathbf{x}) \in \Pi_{\mathscr{S},\varphi_1 \wedge \varphi_2}$ such that $\mathfrak{a} \vDash \pi(\mathbf{x})$ and $s = \mathfrak{a}(\mathbf{x})$.

$$\varphi = EX\varphi_1.\text{ Then}$$
$$s \vDash EX\varphi_1 \Leftrightarrow$$

there exists a successor s' of s in $\mathscr{M}_{\mathscr{S}}$ such that $s' \vDash \varphi_1 \Leftrightarrow$

there exists $i \in \{1, \dots, k\}$ and an assignment $\mathfrak{a}_i$ such that $\mathfrak{a}_i(\delta_i) = 1$, $\mathfrak{a}_i(\delta_j) = 0$ for $j \neq i$, $\mathfrak{a}_i \models C_i(\mathbf{x}, \mathbf{y})$, $\mathfrak{a}_i(\mathbf{x}) = s$ and $\mathfrak{a}_i(\mathbf{y}) = s'$, and there exist $\pi(\mathbf{y}) \in \Pi_{\mathscr{S},\varphi_1}$ and an assignment $\mathfrak{a}'$ such that $\mathfrak{a}' \models \pi(\mathbf{y})$ and $\mathfrak{a}'(\mathbf{y}) = s' \Leftrightarrow$

there exist $\pi(\mathbf{x}) \in \Pi_{\mathscr{S},EX\varphi_1}$ and an assignment $\mathfrak{a}$ such that $\mathfrak{a} \models \pi(\mathbf{x})$ and $\mathfrak{a}(\mathbf{x}) = s$.

$$\varphi = AX\varphi_1. \text{ Then}$$
$$s \models AX\varphi_1 \Leftrightarrow$$

for every successor s_i of s in $\mathscr{M}_{\mathscr{S}}$ it holds that $s_i \models \varphi_1 \Leftrightarrow$

for each $i = 1, \dots, k$, it holds that there exists an assignment $\mathfrak{a}_i$ such that $\mathfrak{a}_i \models C_i(\mathbf{x}, \mathbf{y}_i)$, $\mathfrak{a}_i(\mathbf{x}) = s$, $\mathfrak{a}_i(\mathbf{y}_i) = s_i$, and there exist $\pi_i(\mathbf{y_i}) \in \Pi_{\mathscr{S},\varphi_1}$ and an assignment $\mathfrak{a}'_i$ such that $\mathfrak{a}'_i \models \pi_i(\mathbf{y}_i)$ and $\mathfrak{a}'_i(\mathbf{y}_i) = s_i$

$$\begin{aligned} vars(\pi_i(\mathbf{y}_i)) \cap vars(\pi_j(\mathbf{y}_j)) &= \emptyset \\ vars(C_i(\mathbf{x}, \mathbf{y}_i)) \cap vars(C_j(\mathbf{x}, \mathbf{y}_j)) &= \mathbf{x} \\ vars(C_i(\mathbf{x}, \mathbf{y}_i)) \cap vars(\pi_i(\mathbf{y}_i)) &= \mathbf{y}_j \\ \Leftrightarrow \quad vars(C_i(\mathbf{x}, \mathbf{y}_i)) \cap vars(\pi_j(\mathbf{y}_j)) &= \emptyset \end{aligned}$$

there exist $\pi(\mathbf{x}) \in \Pi_{\mathscr{S},AX\varphi_1}$ and an assignment $\mathfrak{a}$ such that $\mathfrak{a} \models \pi(\mathbf{x})$ and $\mathfrak{a}(\mathbf{x}) = s$. □

We now devise a compositional verification procedure that searches for a feasible MILP in the set of MILPs generated by the compositional encoding. Similarly to the monolithic procedure, we pass the negation of the property to be verified together with a restriction to the initial states of $\mathscr{S}$ to the compositional encoding. If all problems in $\Pi_{\mathscr{S},\mathsf{NNF}(\neg\varphi)\wedge\varphi_I}$ are infeasible, no initial state $s \in I$ and no possible evolution of the system from s could be found that would make $\neg\varphi$ true; therefore, the property is satisfied and the procedure returns `True`. However, if at least one of the programs has a solution, from the solution we can extract the counterexample for the specification in question, so the procedure returns `False`. The procedure is presented in Algorithm 2.

Since the feasibility checks of the generated programs can be executed independently of each other, they naturally lend themselves to parallelisation. The compositional procedure can thus be particularly efficient at finding bugs that can be reached within a few steps along some of the paths from the initial states. This has parallels with bounded model checking [12] since, once a bug has been detected, the whole procedure can be terminated. As we will see in the next section, this is particularly useful when verifying bounded safety.

Algorithm 2 The compositional MILP verification procedure.

```
procedure COMP-VERIFY($\mathscr{S}$, $\varphi$)
    Input: NANES $\mathscr{S} = (Ag, E, I)$; formula $\varphi \in \mathrm{bCTL}_{\mathbb{R}^<}$
    Output: True/False
    $feasible \leftarrow$ False
    $\varphi_I \leftarrow$ Boolean $\mathrm{bCTL}_{\mathbb{R}^\leq}$ formula representing $I$
    $\varphi' \leftarrow \mathsf{NNF}(\neg\varphi) \wedge \varphi_I$
    $\Pi_{\mathscr{S},\varphi'} \leftarrow$ Set of MILPs associated with $\mathscr{S}$ and $\varphi'$
    for $\pi$ in $\Pi_{\mathscr{S},\varphi'}$ do
        $aux \leftarrow \mathsf{MILP\_SOLVER}(\pi)$
        if $aux$ is True then
            $feasible \leftarrow$ True
            break
    return $\neg feasible$
```

6 Computational complexity of the verification problem

In this section we study the complexity of the verification problem for $\mathrm{bCTL}_{\mathbb{R}^<}$. The upper bound follows from the monolithic verification procedure and the lower bound can be obtained by reduction from the validity problem of QBF.

Theorem 3 *Verifying NANES against* $\mathrm{bCTL}_{\mathbb{R}^<}$ *is in coNExpTime and PSpace-hard in combined complexity.*

Proof The coNExpTime upper bound follows from the fact the MILP program in Algorithm 1 takes exponential time to construct and is of exponential size, and that infeasibility of MILP is a coNP-complete problem.

The PSpace lower bound can be shown by adapting the lower bound proof in [34]. The idea is to represent the full binary tree of variable assignments in the temporal model of NANES and then to use a $\mathrm{bCTL}_{\mathbb{R}^<}$-specification to check validity of the QBF.

Let Φ be a QBF of the form $\mathsf{Q}_1 x_1 \dots \mathsf{Q}_m x_m \varphi$, where φ is in 3CNF. We now construct a NANES $\mathscr{S} = ((Act, prot), (S, t_E), I)$ and a $\mathrm{bCTL}_{\mathbb{R}^<}$-specification ψ such that $\mathscr{S} \models \psi$ iff Φ is valid.

Let n be the number of clauses in φ.

- the state space $S \subseteq \mathbb{R}^{m+1}$ is $[-1,1]^m \times [-1,n]$, where in a state $(a_1, \dots, a_{m+1})$, a_i, $i \leq m$, encodes an assignment to the variable x_i with -1 being the value not yet set, and 0 and 1 being False and Truth respectively, and a_{m+1} holds the number of satisfied clauses in φ under the given assignment, or -1 if the assignment has not been set yet.
- the set $Act \subseteq \mathbb{R}^{m+1}$ of actions is $\{(b_1, \dots, b_{m+1}) \big| \exists i, b_j \neq 0 \text{ iff } j = i \text{ and } b_i = 1 \text{ if } i \leq m\}$
- the transition function t_E,
  - given a state $(v_1, \dots, v_{m+1})$ and an action $(0, \dots, 1, \dots, 0)$ with 1 at position $i \leq m$, returns two states: $(v_1, \dots, v_{i-1}, 0, v_{i+1}, \dots, v_{m+1})$ and $(v_1, \dots, v_{i-1}, 1, v_{i+1}, \dots, v_{m+1})$;
  - given a state $(v_1, \dots, v_m, v_{m+1})$ and an action $(0, \dots, 0, v)$ returns state $(v_1, \dots, v_m, v)$.
- the neural agent performs two different tasks depending on the input. Let $(a_1, \dots, a_{m+1}) \in S$. If at least one of $a_1, \dots, a_m$ is -1, the network returns the vector $(0, \dots, 1, \dots, 0)$ with 1 at position i where i is the minimal index with $a_i = -1$. Other-

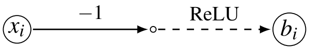
wise, the network computes the value of φ for the assignment given by $a_1, \ldots, a_m$. N is defined in detail below.

- I consists of one initial state $(-1, \ldots, -1)$, and
- the specification property Q is defined as $\mathsf{P}_1 X^1 \ldots \mathsf{P}_m X^1 \big((m+1) = n\big)$, where $\mathsf{P}_i = A$ if $\mathsf{Q}_i = \forall$, $\mathsf{P}_i = E$ if $\mathsf{Q}_i = \exists$.

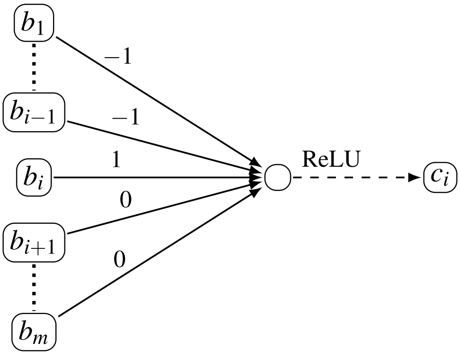
One can show that the agent's protocol function and the environment transition function are PWL. It is straightforward to see that $\mathscr{S}$ and ψ are as required. Observe that PSPACE-hardness holds already for a single initial state, i.e., when I is a singleton set.

We show how to construct the agent's neural network N. We do so by defining a number of gadgets that will constitute N.

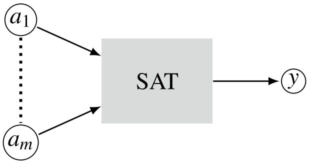
- The *undefined gadget*, given a node $a_i \in \{-1, 0, 1\}$ outputs a node b_i that is 1 if $a_i = -1$ (i.e., the value of variable X_i is not set), and 0 otherwise.



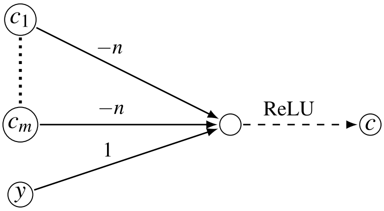
- The *i first undefined gadget*, which given nodes $b_1, \ldots, b_m \in \{0, 1\}$ outputs a node c_i that is 1 if $b_i = 1$ and for $1 \leq j < i$, $b_j = 0$, and 0 otherwise.



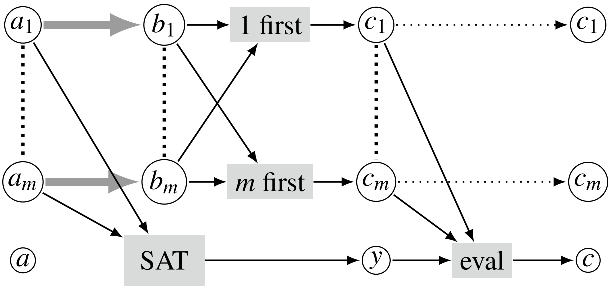
- The *SAT gadget* evaluates φ for the assignment provided by the values of $a_1, \ldots, a_m$, and outputs a node y (see [34] for details). If the assignment is valid (i.e., all a_i are from $\{0, 1\}$, the value of y is n iff it is a satisfying assignment.



- We are going to output y but only if all a_is, $1 \leq i \leq m$, have been assigned a value, and hence c_is are all 0. Otherwise, the value of y will be discarded with the help of the following *evaluation result gadget*.



To conclude with N, the output nodes of N are $c_1, \ldots, c_m, c$. We can schematically depict N as follows:



□

We also show that the complexity of the verification problem is reduced to coNP for the bounded safety fragment of $\mathrm{bCTL}_{\mathbb{R}^<}$.

Corollary 2 *Verifying NANES against bounded safety properties is coNP-complete in combined complexity.*

Proof The upper bound follows from the fact that we can check whether a property $\varphi = AG^k safe$ is not satisfied by $\mathscr{S}$ by guessing an initial state s and a path ρ of length k originating from s, and by verifying that $\rho(i) \not\models safe$ for some $i = 1, \ldots, k$. If such an initial state s exists, then there exists an initial state s' with the same properties of polynomial size. This follows from the encoding into MILP and the fact that if a MILP instance is feasible, there is a solution of polynomial size.

The lower bound can be adapted from the NP lower bound of the satisfiability problem of neural networks properties [34], and holds already for one-step formulae. □

7 Implementation and experiments

We have implemented the verification procedures described in the previous section in an open source toolkit called VENMAS [54]. The tool takes as input a $\mathrm{bCTL}_{\mathbb{R}^<}$ specification φ and a NANES $\mathscr{S}$. The top-level call to the tool returns `True` if φ is satisfied by $\mathscr{S}$, and returns `False` if φ is not satisfied at some initial state of $\mathscr{S}$. In the latter case, a trace in the form of state-action pairs is produced, giving an example run of the system which failed to satisfy the specification.

The user provides a parameter to determine whether the monolithic or compositional procedure with parallel or sequential execution is to be used. For monolithic verification VENMAS follows Algorithm 1. For compositional verification VENMAS performs the computation in line 7 of Algorithm 2 in a *splitting* process that adds subprograms to a *jobs queue*. The computation in line 9 is performed asynchronously across a specified number of *worker* processes (executing on the same machine) retrieving tasks from the jobs queue: one worker process under sequential execution and multiple worker processes under parallel execution (the default in the implementation is 8). The *main* process finishes either when a MILP query terminates with a feasible solution (i.e., a counter-example), or all the jobs return infeasible results, or no result is returned within a given time limit.

In order to produce stronger MILP formulations, both in the case of monolithic and compositional encodings, we compute lower and upper bounds for all state and action variables. These are computed by propagating through the networks the bounds of the input states given by I using symbolic interval propagation.[2] The bounds are used in the Big-M encoding of the ReLU nodes (see Sect. 3.2). For other kind of constraints (e.g., those expressing the transition function), the bounds are propagated using standard interval arithmetic. The bounds of the *EX* successor state variables are taken as the widest bounds among all possible b successors.

We observe that the compositional encoding in Fig. 7 requires computing the whole set $\Pi_{\mathscr{S},\varphi}$ in memory, before the individual jobs can become available to the worker processes. This presents several drawbacks. First, it requires exponential memory and can be infeasible for large networks and big values of temporal depth. Second, the workers are idle at the beginning of the process. Therefore, for the experiments below, we have implemented and used a version of the compositional encoding that accepts the fragment of $\mathrm{bCTL}_{\mathbb{R}^\leq}$ that consists of (arbitrary) disjunctions, conjunctions over atomic propositions, and the existential path quantifier *EX*. The compositional encoding computes the programs in $\Pi_{\mathscr{S},\varphi}$ and adds them to the jobs queue one by one in a depth-first fashion. The worker processes then can start to solve verification queries as soon as individual jobs become available. If a counter-example has been detected in one of the first jobs, the whole verification procedure can terminate without computing all subproblems. Additionally, we integrated a further optimisation whereby we may discard particular jobs by examining the bounds of the state variables when encoding an atomic proposition: if the bounds (e.g., $x_1 \in [-116, -112]$) contradict the atomic formula (e.g, $(1) \geq -100$), then the whole MILP instance is trivially infeasible and no verification job is created. Finally, disjunction of atomic propositions is

[2] Symbolic interval propagation [55] computes the bounds by keeping track of symbolic lower and upper bound equations for each intermediate layer of the network obtained through combination of symbolic interval analysis and linear relaxation of the ReLU constraints. The equations depend on the input layer and thus may be used to capture implicit relationship between nodes of intermediate layers, resulting in reasonably tight bounds that can be computed effectively.

handled as in the monolithic case; this limits the blow-up on the number of MILPs generated whilst not hindering the efficiency of the verification procedure.

The tool is implemented in Python and uses Gurobi ver. 9.1[23] as a back-end to solve the generated MILP problems. To evaluate our tool, we performed experiments on a machine equipped with an Intel Core i7-7700K CPU @ 4.20GHz with 16GB of RAM, running Ubuntu 20.04, kernel version 5.8.

We now describe the experimental results obtained on two scenarios.

7.1 FrozenLake scenario

We started by validating VENMAS on the FrozenLake scenario from Example 1 against the specifications reported in Example 3. We considered the grid world as in Fig. 4: there are two holes – tiles e_3 and e_7, tile e_1 is the initial state, and tile e_9 is the goal.

We trained the agent's neural network using an actor-critic approach [51]. Each training episode ends when the agent reaches the goal or falls in a hole. A reward of 1 is received if the agent reaches the goal, otherwise no reward is given. So, the agent is trained to maximise the chance of reaching the goal, including avoiding the holes. The resulting network is a ReLU-FFNN with 2 hidden layers both consisting of 32 neurons, 9 inputs and 4 outputs. The environment transition function was implemented using appropriate MILP constraints.

To evaluate how apt the agent was at realising safe and successful runs, we verified the FrozenLake system $\mathcal{S}_{\mathsf{fl}}$ against the specifications $\varphi^k_{\mathsf{safe}}$ and $\varphi^k_{\mathsf{succ}}$ previously defined in Example 3, stating, respectively, that all k-bounded runs are safe and all k-bounded runs are safe and successful. To derive more optimal encodings, the specifications were equivalently formulated to minimise the number of AX operators:

$$\varphi^k_{\mathsf{safe}} = \underbrace{AX(\mathtt{ha} \wedge AX(\mathtt{ha} \wedge \cdots (\mathtt{ha} \wedge AX\mathtt{ha})))}_{k\ AX\text{'s}},$$

$$\varphi^k_{\mathsf{succ}} = \underbrace{AX(\mathtt{ha} \wedge AX(\mathtt{ha} \wedge \cdots (\mathtt{ha} \wedge AX\mathtt{goalreached})))}_{k\ AX\text{'s}},$$

The experiments showed that the agent was always able to avoid holes, hence to realise safe runs; however the agent was shown to be unable to ensure that the goal was reached in a given number of time steps.

Table 1 reports the time (in seconds) taken to resolve the specifications $\varphi^k_{\mathsf{safe}}$ and $\varphi^k_{\mathsf{succ}}$ for $k \in \{1, \dots, 10\}$ for each of the execution modes. The results for the monolithic procedure are denoted Monolithic, and the results for the compositional procedure with parallel and sequential execution are denoted Comp-Par and Comp-Seq, respectively. All cases use a fixed timeout of one hour. Here we see that the compositional procedure, regardless of the execution mode, was very efficient both at finding counter-examples to $\varphi^k_{\mathsf{succ}}$ and at proving that $\varphi^k_{\mathsf{safe}}$ is satisfied. There was no difference between the sequential and parallel executions when verifying $\varphi^k_{\mathsf{safe}}$ because all jobs have been discarded by the splitting process.

The monolithic procedure also managed to locate counter-examples quickly. As for proving safety, it was significantly slower than the compositional procedure for $k \geq 6$. The main reason for this is that it produced very loose MILP formulations due to over-approximated bounds. Note that in this scenario we start from a single state; so for every program in the compositional encoding we are able to compute each next state exactly. However, in the monolithic procedure we are forced to over-approximate the bounds of the

Table 1 Verification times for the $\mathscr{L}$ properties $\varphi^k_{\mathsf{safe}}$ and $\varphi^k_{\mathsf{succ}}$ for different values of k on the FrozenLake scenario encoded as a NANES

k	Monolithic		Comp-Seq		Comp-Par	
	$\varphi^k_{\mathsf{safe}}$	$\varphi^k_{\mathsf{succ}}$	$\varphi^k_{\mathsf{safe}}$	$\varphi^k_{\mathsf{succ}}$	$\varphi^k_{\mathsf{safe}}$	$\varphi^k_{\mathsf{succ}}$
1	0.04	*0.04*	0.03	*0.06*	0.04	*0.07*
2	0.07	*0.08*	0.04	*0.09*	0.07	*0.12*
3	0.53	*0.24*	0.14	*0.17*	0.12	*0.24*
4	1.67	*0.58*	0.18	*0.25*	0.19	*0.30*
5	9.50	*0.97*	0.41	*0.26*	0.43	*0.35*
6	241.41	*2.75*	1.16	*0.29*	1.16	*0.35*
7	915.34	*8.61*	3.37	*0.32*	3.41	*0.38*
8	3476.08	*8.70*	10.33	*0.35*	10.30	*0.42*
9	–	*5.29*	32.50	*0.36*	32.68	*0.46*
10	–	*30.97*	111.22	*0.38*	112.87	*0.46*

Italicized cells indicate a `False` result, otherwise a `True` result. A hyphen '–' represents a one hour timeout

state variables $\mathbf{y}$ when encoding $EX\varphi$ since we do not know in advance which of the b successors they refer to. This difficulty is further exacerbated by the discrete nature of the state space in this scenario. To see this, assume that we are in the state e_5, the chosen direction is down, and $\mathbf{x}$ has exact bounds (0, 0, 0, 0, 1, 0, 0, 0, 0). Then the possible successors are e_8, e_4 and e_6, and the bounds become very large: the upper bounds of the state variables $\mathbf{y}$ become (0, 0, 0, 1, 0, 1, 0, 1, 0) and the lower bounds all zeros. These bounds are further loosened as they are propagated through the network resulting in MILP formulations that are hard to solve.

7.2 The aircraft collision avoidance system VerticalCAS

For the second set of experiments, we consider a scenario involving two aircraft, the *ownship* and the *intruder*, where the ownship is equipped with a collision avoidance system VerticalCAS [32]. The intruder is assumed to follow a constant horizontal trajectory. Every second VerticalCAS issues vertical climbrate advisories to the ownship pilot. This is to avoid a *near mid-air collision (NMAC)*, a region where the ownship and intruder are separated by less than 100ft vertically and 500ft horizontally. The possible advisories are:

(1) COC: Clear Of Conflict.
(2) DNC: Do Not Climb.
(3) DND: Do Not Descend.
(4) DES1500: Descend at least 1500 ft/s.
(5) CL1500: Climb at least 1500 ft/s.
(6) SDES1500: Strengthen Descent to at least 1500 ft/s.
(7) SCL1500: Strengthen Climb to at least 1500 ft/s.
(8) SDES2500: Strengthen Descent to at least 2500 ft/s.
(9) SCL2500: Strengthen Climb to at least 2500 ft/s.

The advisories instruct the pilot to accelerate until the vertical climbrate of the ownship complies with the advisory. For some advisories, e.g. DND, the pilot can choose any acceleration in $[\frac{g}{4}, \frac{g}{3}]$, where g represents the gravitational constant 32.2 ft/s^2. In what follows,

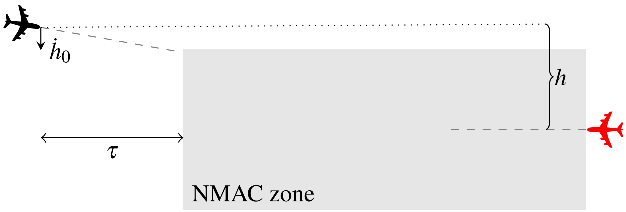


Fig. 8 VerticalCAS encounter geometry

with slight abuse of notation, we refer to advisories using both their human readable and the corresponding numeric notation.

We hereafter denote by $[m]$ the set $\{1, \dots, m\}$. Consider the set $S_{\mathsf{vcas}} = [-3000, 3000] \times [-2500, 2500] \times [0, 40] \times [9]$. A tuple $(h, \dot{h}_0, \tau, \mathrm{adv}) \in S_{\mathsf{vcas}}$ describes an ownship–intruder *encounter*, where:

1) h (ft) is the intruder altitude relative to ownship.
2) $\dot{h}_0$ (ft/s) is the ownship vertical climbrate.
2) τ (s) is the time to loss of horizontal separation.
4) adv is the previous advisory issued by VerticalCAS.

Figure 8 illustrates the vertical geometry of the encounter, which is given by h and $\dot{h}_0$, and the time τ until the ownship (black) and intruder (red) are no longer horizontally separated.

The VerticalCAS system is composed of nine ReLU-FFNNs $F = \left\{\left(f_i : \mathbb{R}^3 \to \mathbb{R}^9\right) : i \in [9]\right\}$, one for each advisory, with three inputs $(h, \dot{h}_0, \tau)$, five fully-connected hidden layers of 20 units each, and nine outputs representing the score of each possible advisory.

7.2.1 NANES encoding and specification

We now formalise the VERTICALCAS scenario as a NANES $\mathscr{S}_{\mathsf{vcas}} = (Ag_{\mathsf{vcas}}, E_{\mathsf{vcas}}, I_{\mathsf{vcas}})$. We model VerticalCAS as the neural agent $Ag_{\mathsf{vcas}} = (Act_{\mathsf{vcas}}, prot_{\mathsf{vcas}})$ with the set of actions $Act_{\mathsf{vcas}} = [9]$ and the protocol function producing an action corresponding to the highest-scoring advisory formally defined as $prot_{\mathsf{vcas}}(s) = \arg\max\left(\mathsf{apply}(\mathsf{select}(s), s)\right)$, where for a state $s = (h, \dot{h}_0, \tau, \mathrm{adv}) \in S_{\mathsf{vcas}}$:

- $\mathsf{select} : S_{\mathsf{vcas}} \to F$ selects the neural network corresponding to the previous advisory adv, $\mathsf{select}(s) = f_{\mathrm{adv}}$;
- $\mathsf{apply} : F \times \mathbb{R}^4 \to \mathbb{R}^9$ computes the output of a neural network for a given state, $\mathsf{apply}(f, s) = f(h, \dot{h}_0, \tau)$,
- $\arg\max : \mathbb{R}^9 \to [9]$ returns the index of the score with highest value from a neural network's output.

Since each of the above functions and the ReLU-FFNNs are PWL, the composition *prot* is also PWL.

We model the ownship pilot's non-deterministic behaviour in the environment of $\mathscr{S}_{\mathsf{vcas}}$, defined as $E_{\mathsf{vcas}} = (S_{\mathsf{vcas}}, t_{E_{\mathsf{vcas}}})$. Thus, the environment transition function $t_{E_{\mathsf{vcas}}}$ "chooses" an acceleration and determines the next state of the environment through the state transition dynamics.

The acceleration chosen by the pilot depends on the issued advisory adv' and the current vertical climbrate $\dot{h}_0$ of the ownship. We bound the number of possible successor states of $t_{E_{\mathrm{vcas}}}$ by 3, that is $b = 3$, so the set of next possible accelerations is $Acc^{\dot{h}_0}_{\mathrm{adv}'} = \{\ddot{h}_0^{(1)}, \ddot{h}_0^{(2)}, \ddot{h}_0^{(3)}\}$ defined as follows. If the current vertical climbrate $\dot{h}_0$ of the ownship is *compliant* with the advisory, the pilot maintains a constant climbrate, i.e., $\ddot{h}_0^{(i)} = 0$ for $i \in [b]$. Otherwise, the pilot chooses acceleration from the continuous interval defined for each advisory. We discretise the set of possible accelerations into b equally spaced cells, so for instance, $Acc^{\dot{h}_0}_{\mathrm{DND}} = \{\frac{g}{4}, \frac{7g}{24}, \frac{g}{3}\}$.

Given the current state $s \in S_{\mathrm{vcas}}$, the issued advisory $\mathrm{adv}' = prot_{\mathrm{vcas}}(s)$, and the set of next possible accelerations $Acc^{\dot{h}_0}_{\mathrm{adv}'} = \{\ddot{h}_0^{(i)} : i \in [b]\}$, we define each of the transition functions $t_1, \dots, t_b$ for $t_{E_{\mathrm{vcas}}}$ as:

$$t_i\left(\begin{bmatrix} h \\ \dot{h}_0 \\ \tau \\ \mathrm{adv} \end{bmatrix}, \mathrm{adv}'\right) = \begin{bmatrix} h - \dot{h}_0 \Delta\tau - 0.5\ddot{h}_0^{(i)} \Delta\tau^2 \\ \dot{h}_0 + \ddot{h}_0^{(i)} \Delta\tau \\ \tau - \Delta\tau \\ \mathrm{adv}' \end{bmatrix},$$

where $\Delta\tau = 1$ and $i \in [b]$. Thus, each t_i simulates a possible choice of acceleration by the pilot *and* computes the next state by taking into account the state transition dynamics.

The set I_{vcas} of initial states is defined as

$$[-133, -129] \times \dot{H}_0 \times \{25\} \times \{\mathrm{COC}\},$$

where $\dot{H}_0 = \{-19.5 - 3k \,|\, k = 0, \dots, 8\} \cup \{-39, -39.5, -40\}$ the set of all initial climbrates. This is a potentially risky encounter with the intruder initially below the ownship, but with the ownship descending towards the intruder.

We are interested in checking whether by following the advisories issued by VerticalCAS and independently of the acceleration chosen by the pilot, the ownship can manage to stay outside of the *unsafe region* ($|h| \leq 100$), entering which may potentially lead to an NMAC for small values of τ. So we consider the safety specifications

$$\varphi^k = AX^k\,((1) > 100 \ \vee \ (1) < -100)$$

for various values of k. Recall that the term (1) represents the first component of the state s and so refers to h, the intruder altitude relative to ownship. Thus, the formula φ^k states that in every evolution of the system starting from every initial state in I_{vcas} after k time steps the absolute value of vertical separation is greater than 100 ft.

7.2.2 Implementation and experiments

Our implementation of VerticalCAS system aims to compute tightest possible bounds for all states and intermediate variables in an effort to produce efficient MILP encodings.

We implemented the agent as a combination of custom MILP constraints and of NN MILP encodings. Note that in the worst case we need to choose between 9 networks to compute the action (next advisory), requiring 9 additional binary variables (similarly to encoding *EX*) and the constraints for each of the 9 networks. We keep the number of MILP constraints as low as possible, by avoiding to include the encodings of the networks f_{adv} for advisory values that lie outside of the current bounds of the (previous) advisory variable. For instance, if these bounds identify advisory as being COC (i.e., lower and upper bounds are 1), then only the encoding of f_1 is included. This also enables computing tighter bounds for the new advisory.

The environment is implemented via a set of custom MILP constraints. Given a new advisory by the agent, we need to choose the acceleration depending on 9 possible values of the advisory. Again, we compute tight acceleration bounds by considering the bounds of the new advisory. In particular, when the latter are exact (i.e., the exact value of the advisory is known), the former are exact as well. Tight acceleration bounds allow for computing tight bounds for the next state, and so on.

We verified the VerticalCAS system $\mathscr{S}_{\mathsf{vcas}}$ against the specifications φ^k for various values of k. The experiments showed that for high values of descent rate the ownship enters the unsafe region for a number of steps, eventually managing to recover. As the descent rate decreases, the time spent in the unsafe region decreases, until for the lowest value of -19.5 where the ownship remains safe for the entire period. For instance, the trace produced by VENMAS for $\dot{h}_0 = -22.5$ and $k = 3$ shows that the agent issues the advisory CL1500 at each time step, thereby causing the pilot to accelerate at $\frac{g}{4}$ft/s^2 so as to climb to avoid collision with the intruder. The descent rate was not reduced quickly enough to avoid the unsafe state $(h, \dot{h}_0, \tau, \text{adv}) = (-97.725,\ 1.65,\ 22,\ \text{CL1500})$ being reached by the third timestep.

Table 2 reports the performance of the tool in terms of time (in seconds) taken to resolve the specification φ^k for $k \in \{1, \dots, 10\}$ with initial climbrates $\dot{h}_0 \in \dot{H}_0$ for each of the execution modes. For all cases we use a fixed timeout of one hour. Here we see that the monolithic procedure is the most performant method, both for proving safety and for finding counter-examples. We attribute this to the fact that, unlike in the FrozenLake scenario, here it was possible to compute tight bounds for the state variables even after 10 time steps, resulting in tight MILP formulations whose (in)feasibility can be solved efficiently by a MILP solver. The compositional procedure was penalised as it had to construct an exponential number of programs and analyse each of them for the configurations where the property was satisfied. As before, there is no difference between sequential and parallel executions when all created subproblems have been discarded early.

We have also identified several more challenging configurations that are likely to lie close to the boundary between the safe and unsafe initial positions, such as for $\dot{h}_0 \in \{-39, -39.5, -40, -40.5\}$ for $k \in \{9, 10\}$. Because of the uncertainty, the bounds become looser. As a result, not all problems are discarded during the compositional encoding and the workers are assigned MILP problems; in this case the parallel execution is approximately twice as fast as the sequential execution. Instead, the monolithic procedure required more branch and bound iterations to find a feasible assignment or to rule out its existence. Among these initial states we also found few cases where compositional procedure was more efficient than the monolithic one ($\dot{h}_0 = -39, k = 9$; $\dot{h}_0 = -39.5, k = 10$; and $\dot{h}_0 = -40.5, k = 9$).

Lastly, we note that we used double-precision floating point numbers for representing real values. Gurobi, the back-end MILP solver that we used, uses a default tolerance of 10^{-6}, representing the amount of numerical error allowed on a constraint while still considering it “satisfied”. We relied on Gurobi for dealing with any further numerical issues. Finally, note that the encoding here presented is more efficient than [3], which does not compute variable bounds for their MILP encoding.

In terms of comparisons, we are unable to present an evaluation with other tools because, as far as we are aware, no other tool supports branching models and CTL specifications as we do here.

Table 2 Verification times for a VerticalCAS system against the property φ^k for different values of k and $\dot{h}_0$

MONOLITHIC												
$\dot{h}_0$	−19.5	−22.5	−25.5	−28.5	−31.5	−34.5	−37.5	−39.0	−39.5	−40.0	−40.5	−43.5
φ^1	0.05	0.06	0.04	0.05	0.04	*0.05*	*0.05*	*0.04*	*0.04*	*0.06*	*0.05*	*0.06*
φ^2	0.10	0.10	*0.12*	*0.10*	*0.10*	*0.10*	*0.11*	*0.11*	*0.10*	*0.11*	*0.10*	*0.11*
φ^3	0.12	*0.13*	*0.13*	*0.13*	*0.13*	*0.13*	*0.13*	*0.13*	*0.14*	*0.12*	*0.13*	*0.13*
φ^4	0.14	0.14	*0.15*	*0.15*	*0.15*	*0.15*	*0.15*	*0.15*	*0.15*	*0.15*	*0.16*	*0.15*
φ^5	0.16	0.16	0.19	*0.17*	*0.18*	*0.17*	*0.17*	*0.18*	*0.18*	*0.17*	*0.18*	*0.18*
φ^6	0.18	0.18	0.18	0.19	*0.20*	*0.23*	*0.20*	*0.20*	*0.20*	*0.19*	*0.19*	*0.22*
φ^7	0.19	0.21	0.19	0.20	0.19	*0.26*	*0.25*	*0.25*	*0.24*	*0.24*	*0.28*	*0.46*
φ^8	0.22	0.22	0.25	0.25	0.22	0.23	*0.39*	*0.51*	*0.50*	*0.97*	*0.99*	*1.21*
φ^9	0.27	0.24	0.45	0.47	0.23	0.24	0.75	227.91	65.42	122.29	*59.28*	*8.37*
φ^{10}	0.48	0.36	0.62	1.95	0.26	0.26	2.66	67.79	1302.12	1761.43	1552.48	–
COMP-SEQ												
$\dot{h}_0$	−19.5	−22.5	−25.5	−28.5	−31.5	−34.5	−37.5	−39.0	−39.5	−40.0	−40.5	−43.5
φ^1	0.04	0.03	0.04	0.04	0.04	*0.08*	*0.08*	*0.07*	*0.08*	*0.08*	*0.08*	*0.07*
φ^2	0.10	0.10	*0.23*	*0.23*	*0.23*	*0.23*	*0.22*	*0.23*	*0.22*	*0.22*	*0.23*	*0.23*
φ^3	0.13	*0.25*	*0.25*	*0.27*	*0.25*	*0.25*	*0.25*	*0.26*	*0.25*	*0.25*	*0.27*	*0.29*
φ^4	0.26	0.25	*0.28*	*0.28*	*0.28*	*0.29*	*0.29*	*0.31*	*0.28*	*0.28*	*0.29*	*0.29*
φ^5	0.58	0.58	0.58	*0.32*	*0.33*	*0.32*	*0.32*	*0.33*	*0.32*	*0.32*	*0.32*	*0.32*
φ^6	1.63	1.72	1.64	1.64	*0.37*	*0.38*	*0.37*	*0.36*	*0.35*	*0.35*	*0.38*	*0.38*
φ^7	4.76	4.75	4.76	4.78	4.75	*0.40*	*0.38*	*0.38*	*0.38*	*0.38*	*0.38*	*0.40*
φ^8	14.67	14.82	14.73	14.88	14.70	14.73	*0.45*	*0.42*	*0.44*	*0.44*	*0.42*	*0.43*
φ^9	47.51	47.51	47.36	48.68	47.44	47.63	48.00	170.23	295.32	463.64	*0.50*	*0.46*
φ^{10}	169.57	169.76	169.93	169.48	169.24	169.67	178.62	987.33	1854.97	3183.62	–	–
COMP-PAR												
$\dot{h}_0$	−19.5	−22.5	−25.5	−28.5	−31.5	−34.5	−37.5	−39.0	−39.5	−40.0	−40.5	−43.5

Table 2 (continued)

φ^1	0.05	0.06	0.05	0.07	0.06	*0.08*	*0.08*	*0.09*	*0.09*	*0.09*	*0.10*	*0.11*
φ^2	0.12	0.11	*0.29*	*0.32*	*0.32*	*0.34*	*0.31*	*0.33*	*0.30*	*0.29*	*0.31*	*0.30*
φ^3	0.15	*0.28*	*0.31*	*0.31*	*0.34*	*0.33*	*0.31*	*0.31*	*0.29*	*0.33*	*0.39*	*0.32*
φ^4	0.27	0.27	*0.35*	*0.34*	*0.41*	*0.41*	*0.35*	*0.38*	*0.36*	*0.38*	*0.39*	*0.45*
φ^5	0.60	0.64	0.60	*0.43*	*0.34*	*0.40*	*0.36*	*0.43*	*0.38*	*0.42*	*0.47*	*0.35*
φ^6	1.67	1.66	1.67	1.78	*0.48*	*0.38*	*0.41*	*0.49*	*0.43*	*0.38*	*0.40*	*0.51*
φ^7	4.84	4.79	4.73	4.80	4.80	*0.46*	*0.51*	*0.45*	*0.53*	*0.43*	*0.41*	*0.51*
φ^8	14.71	14.75	14.83	14.68	14.76	14.79	*0.48*	*0.44*	*0.45*	*0.49*	*0.46*	*0.45*
φ^9	47.81	47.22	47.45	47.41	47.32	47.67	48.50	97.99	153.44	255.59	*0.47*	*0.48*
φ^{10}	171.53	171.06	175.13	171.55	170.89	172.91	180.81	563.37	1109.42	2167.44	3435.29	–

Italicized cells indicate a `False` result, otherwise a `True` result. We use dashes ‘–’ to indicate a one hour timeout

8 Conclusions

As we argued in Sect. 1, forthcoming autonomous systems will make greater use of machine learning methods; therefore there is an urgent need to develop techniques aimed at providing guarantees on the resulting behaviour of such systems. While the benefits of formal methods have long been recognised, and they have found large adoption in safety-critical systems as well as in industrial-scale software, there have been few efforts to introduce verification techniques for systems driven by neural networks.

In this paper we defined a system composed of a neural agent driven by deep feed-forward neural networks interacting with a non-deterministic environment. The resulting system displays branching evolutions. We defined and studied the resulting verification problem. While the problem is undecidable for full reachability, we isolated a fragment of the temporal language and showed that its corresponding verification problem is in coNExpTime. We developed and reported on a toolkit which includes a novel parallel algorithm to verify temporal properties of the complex environment defined in the VerticalCAS scenario. As demonstrated, while the parallel algorithm remains complete, it offers considerable advantages over its sequential counterpart when searching for counterexamples to bounded safety specifications in concrete examples.

In future work we plan to tackle scalability issues by developing alternative encodings to the ones here presented.

Acknowledgements This work was partly funded by DARPA under the Assured Autonomy programme (FA8750-18-C-0095). Alessio Lomuscio is supported by a Royal Academy of Engineering Chair in Emerging Technologies.

References

1. Akintunde, M., Botoeva, E., Kouvaros, P., & Lomuscio, A. (2020). Formal verification of neural agents in non-deterministic environments. In *Proceedings of the 19th international conference on autonomous agents and multi-agent systems (AAMAS20)* (pp. 25–33). IFAAMAS.
2. Akintunde, M., Kevorchian, A., Lomuscio, A., & Pirovano, E. (2019). Verification of RNN-based neural agent-environment systems. In *Proceedings of the 33rd AAAI conference on artificial intelligence (AAAI19)* (pp. 6006–6013). AAAI Press.
3. Akintunde, M., Lomuscio, A., Maganti, L., & Pirovano, E. (2018). Reachability analysis for neural agent-environment systems. In *Proceedings of the 16th international conference on principles of knowledge representation and reasoning (KR18)* (pp. 184–193). AAAI Press.
4. Anderson, R., Huchette, J., Ma, W., Tjandraatmadja, C., & Vielma, J. (2020). Strong mixed-integer programming formulations for trained neural networks. *Mathematical Programming* pp. 1–37.
5. Bastani, O., Ioannou, Y., Lampropoulos, L., Vytiniotis, D., Nori, A., & Criminisi, A. (2016). Measuring neural net robustness with constraints. In *Proceedings of the 30th international conference on neural information processing systems (NIPS16)* (pp. 2613–2621).

6. Battern, B., Kouvaros, P., Lomuscio, A., & Y. Zheng. (2021). Efficient neural network verification via layer-based semidefinite relaxations and linear cuts. In *Proceedings of the 30th international joint conference on artificial intelligence (IJCAI21)*. To Appear. ijcai.org.
7. Biere, A., Cimatti, A., Clarke, E., Strichman, O., & Zhu, Y. (2003). Bounded model checking. In *Highly dependable software. Advances in computers* (Vol. 58). Academic Press. Pre-print.
8. Bordini, R., Fisher, M., Visser, W., & Wooldridge, M. (2006). Verifying multi-agent programs by model checking. *Autonomous Agents and Multi-Agent Systems, 12*(2), 239–256.
9. Botoeva, E., Kouvaros, P., Kronqvist, J., Lomuscio, A., & Misener, R. (2020). Efficient verification of neural networks via dependency analysis. In *Proceedings of the 34th AAAI conference on artificial intelligence (AAAI20)* (pp. 3291–3299). AAAI Press.
10. Bunel, R., Lu, J., Turkaslan, I., Kohli, P., Torr, P., & Mudigonda, P. (2020). Branch and bound for piecewise linear neural network verification. *Journal of Machine Learning Research, 21*(42), 1–39.
11. Cheng, C., Nührenberg, G., & Ruess, H. (2017). Maximum resilience of artificial neural networks. In *International symposium on automated technology for verification and analysis (ATVA17)* (pp. 251–268). Springer.
12. Clarke, E., Biere, A., Raimi, R., & Zhu, Y. (2001). Bounded model checking using satisfiability solving. *Formal Methods in System Design, 19*(1), 7–34.
13. Clarke, E., Grumberg, O., & Peled, D. (1999). *Model checking*. The MIT Press.
14. Doan, T., Yao, Y., Alechina, N., & Logan, B. (2014). Verifying heterogeneous multi-agent programs. In *Proceedings of the 13th international conference on autonomous agents and multi-agent systems (AAMAS14)* (pp. 149–156).
15. Dutta, S., Chen, X., & Sankaranarayanan, S. (2019). Reachability analysis for neural feedback systems using regressive polynomial rule inference. In *Proceedings of the 22nd ACM international conference on hybrid systems: Computation and control (HSCC19)* (pp. 157–168). ACM.
16. Dvijotham, K., Stanforth, R., Gowal, S., Mann, T., & Kohli, P. (2018). A dual approach to scalable verification of deep networks. arXiv preprint arXiv:1803.06567.
17. Ehlers, R. (2017). Formal verification of piece-wise linear feed-forward neural networks. In *Proceedings of the 15th international symposium on automated technology for verification and analysis (ATVA17). Lecture notes in computer science* (Vol. 10482, pp. 269–286). Springer.
18. Emerson, E., Mok, A., Sistla, A., & Srinivasan, J. (1992). Quantitative temporal reasoning. *Real-Time Systems, 4*(4), 331–352.
19. Fagin, R., Halpern, J., Moses, Y., & Vardi, M. (1995). *Reasoning about knowledge*. MIT Press.
20. Gammie, P., & van der Meyden, R. (2004). MCK: Model checking the logic of knowledge. In *Proceedings of 16th international conference on computer aided verification (CAV04). Lecture notes in computer science* (Vol. 3114, pp. 479–483). Springer.
21. Goodfellow, A., Bengio, Y., & Courville, A. (2016). *Deep learning* (Vol. 1). Cambridge: MIT press.
22. Griva, I., Nash, S., & Sofer, A. (2009). *Linear and nonlinear optimization* (Vol. 108). Siam.
23. Gu, Z., Rothberg, E., & Bixby, R. (2020). Gurobi optimizer reference manual. http://www.gurobi.com
24. Guidotti, R., Monreale, A., Ruggieri, S., Turini, F., Giannotti, F., & Pedreschi, D. (2019). A survey of methods for explaining black box models. *ACM Computing Surveys, 51*(5), 93:1-93:42.
25. Haykin, S. (1999). *Neural networks: A comprehensive foundation*. Prentice Hall.
26. Henriksen, P., & Lomuscio, A. (2020). Efficient neural network verification via adaptive refinement and adversarial search. In *Proceedings of the 24th European conference on artificial intelligence (ECAI20)* (pp. 2513–2520). IOS Press.
27. Henriksen, P., & Lomuscio, A. (2021). DEEPSPLIT: An efficient splitting method for neural network verification via indirect effect analysis. In *Proceedings of the 30th international joint conference on artificial intelligence (IJCAI21)*. To Appear. ijcai.org.
28. Huang, C., Fan, J., Li, W., Chen, X., & Zhu, Q. (2019). ReachNN: Reachability analysis of neural-network controlled systems. *ACM Transactions on Embedded Computing Systems (TECS), 18*(106), 1–22.
29. Huang, X., Kroening, D., Ruan, W., Sharp, J., Sun, Y., Thamo, E., Wu, M., & Yi, X. (2020). A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability. *Computer Science Reviews, 37*, 100270.
30. Hunt, K., Sbarbaro, D., Zbikowski, R., & Gawthrop, P. (1992). Neural networks for control systems: A survey. *Automatica, 28*(6), 1083–1112.
31. Ivanov, R., Weimer, J., Alur, R., Pappas, G., & Lee, I. (2019). Verisig: Verifying safety properties of hybrid systems with neural network controllers. In *Proceedings of the 22nd ACM international conference on hybrid systems: Computation and control (HSCC19)* (pp. 169–178).

32. Julian, K., & Kochenderfer, M. (2019). A reachability method for verifying dynamical systems with deep neural network controllers. arXiv preprint arXiv:1903.00520.
33. Julian, K., Lopez, J., Brush, J., Owen, M., & Kochenderfer, M. (2016). Policy compression for aircraft collision avoidance systems. In *Proceedings of the 35th digital avionics systems conference (DASC16)* (pp. 1–10).
34. Katz, G., Barrett, C., Dill, D., Julian, K., & Kochenderfer, M. (2017). Reluplex: An efficient SMT solver for verifying deep neural networks. In *Proceedings of the 29th international conference on computer aided verification (CAV17). Lecture notes in computer science* (Vol. 10426, pp. 97–117). Springer.
35. Kouvaros, P., & Lomuscio, A. (2016). Parameterised verification for multi-agent systems. *Artificial Intelligence, 234*, 152–189.
36. Kouvaros, P., & Lomuscio, A. (2018). Formal verification of cnn-based perception systems. arXiv preprint arXiv:1811.11373.
37. Kouvaros, P., & Lomuscio, A. (2021). Towards scalable complete verification of relu neural networks via dependency-based branching. In *Proceedings of the 30th international joint conference on artificial intelligence (IJCAI21)*. To Appear. ijcai.org.
38. Krizhevsky, A., Sutskever, I., & Hinton, G. (2012). Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th international conference on neural information processing systems (NIPS12)* (pp. 1097–1105). Curran Associates, Inc.
39. Liu, C., Arnon, T., Lazaru, C., Barrett, C., & Kochenderfer, M. (2019). Algorithms for verifying deep neural networks. arXiv preprint arXiv:1903.06758.
40. Lomuscio, A., & Maganti, L. (2017). An approach to reachability analysis for feed-forward relu neural networks. arXiv:1706.07351.
41. Lomuscio, A., Qu, H., & Raimondi, F. (2017). MCMAS: A model checker for the verification of multi-agent systems. *Software Tools for Technology Transfer, 19*(1), 9–30.
42. Maes, P. (1993). Modeling adaptive autonomous agents. *Artificial Life, 1*(1–2), 135–162.
43. Nair, V., & Hinton, G. (2010). Rectified linear units improve restricted Boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML10)* (pp. 807–814). Omnipress.
44. Narodytska, N. (2018). Formal analysis of deep binarized neural networks. In *Proceedings of the 27th international joint conference on artificial intelligence, (IJCAI18)* (pp. 5692–5696).
45. OpenAI: Frozenlake-v0. https://gym.openai.com/envs/FrozenLake-v0/ (2019).
46. Papadimitriou, C. H., & Steiglitz, K. (1982). *Combinatorial optimization: Algorithms and omplexity*. Prentice-Hall Inc.
47. Penczek, W., & Lomuscio, A. (2003). Verifying epistemic properties of multi-agent systems via bounded model checking. *Fundamenta Informaticae, 55*(2), 167–185.
48. Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR16)* (pp. 779–788).
49. Siegelmann, H., & Sontag, E. (1995). On the computational power of neural nets. *Journal of Computer and System Sciences, 50*(1), 132–150.
50. Singh, G., Gehr, T., Püschel, M., & Vechev, P. (2019). An abstract domain for certifying neural networks. In *ACM on programming languages* (Vol. 3, pp. 1–30). ACM Press.
51. Sutton, R., & Barto, A. (1998). *Reinforcement learning: An introduction*. MIT Press.
52. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., & Fergus, R. (2014). Intriguing properties of neural networks. In *Proceedings of the 2nd international conference on learning representations (ICLR14)*.
53. Tjeng, V., Xiao, K., & Tedrake, R. (2019). Evaluating robustness of neural networks with mixed integer programming. In *Proceedings of the 7th international conference on learning representations (ICLR19)*.
54. VENMAS: VErification of Neural Multi-Agent Systems. https://vas.doc.ic.ac.uk/software/neural (2020).
55. Wang, S., Pei, K., Whitehouse, J., Yang, J., & Jana, S. (2018). Efficient formal safety analysis of neural networks. In *Advances in neural information processing systems (NeurIPS18)* (pp. 6367–6377).
56. Winston, W. (1987). *Operations research: Applications and algorithms*. Duxbury Press.
57. Xiang, W., H., Rosenfeld, J., & Johnson, T. (2018). Reachable set estimation and safety verification for piecewise linear systems with neural network controllers. In *2018 Annual American control conference (ACC)* (pp. 1574–1579). AACC.