



AFRL-RI-RS-TR-2023-152

VENUS: FORMAL VERIFICATION FOR NEURAL SYSTEMS

IMPERIAL COLLEGE OF SCIENCE TECHNOLOGY & MEDICINE

JULY 2023

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (<http://www.dtic.mil>).

AFRL-RI-RS-TR-2023-152 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /

WILLIAM E. MCKEEVER
Work Unit Manager

/ S /

GREGORY J. HADYNSKI
Assistant Technical Advisor
Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

1. REPORT DATE		2. REPORT TYPE		3. DATES COVERED	
JULY 2023		FINAL TECHNICAL REPORT		START DATE	END DATE
				JULY 2018	JANUARY 2023
4. TITLE AND SUBTITLE					
VENUS: FORMAL VERIFICATION FOR NEURAL SYSTEMS					
5a. CONTRACT NUMBER		5b. GRANT NUMBER		5c. PROGRAM ELEMENT NUMBER	
FA8750-18-C-0095		N/A		62303E	
5d. PROJECT NUMBER		5e. TASK NUMBER		5f. WORK UNIT NUMBER	
				R2H8	
6. AUTHOR(S)					
Alessio Lomuscio					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)					8. PERFORMING ORGANIZATION REPORT NUMBER
IMPERIAL COLLEGE OF SCIENCE TECHNOLOGY & MEDICINE Exhibition Road London SW7 2AZ United Kingdom					
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSOR/MONITOR'S ACRONYM(S)		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
Air Force Research Laboratory/RITA 525 Brooks Road Rome NY 13441-4505			DARPA I2O 675 North Randolph S Arlington VA 22203		AFRL-RI-RS-TR-2023-152
12. DISTRIBUTION/AVAILABILITY STATEMENT					
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT					
The overarching objective of the project "Venus: Formal Verification of Neural Systems" concerned the development of the theoretical underpinnings, methods, and tools for the formal analysis of learning-enabled cyber-physical systems (LE-CPS) and their assessment on usecases of industrial relevance within the "Assured Autonomy" project. Specifically the effort was aimed at: 1. The formalization of the problem of formal verification for open-loop and closed-loop LE-CPS both in terms of decision problem and in terms of specifications to be verified. 2. The development of algorithms and methods for the scalable verification of LE-CPS. 3. The open-source development and release of a toolkit for the methods of point 2. 4. The application of the tool in usecases arising from Technical Area (TA)4 performers.					
15. SUBJECT TERMS					
Formal Verification of Machine Learning, Assured Autonomy, verifying Neural Networks, Assuring Learning-enabled Components					
16. SECURITY CLASSIFICATION OF:				17. LIMITATION OF ABSTRACT	
a. REPORT		b. ABSTRACT		c. THIS PAGE	
U		U		U	
				SAR	
				113	
19a. NAME OF RESPONSIBLE PERSON					19b. PHONE NUMBER (Include area code)
WILLIAM E. MCKEEVER					N/A

TABLE OF CONTENTS

List of Figures	ii
1.0 SUMMARY	1
2.0 INTRODUCTION	2
3.0 METHODS, ASSUMPTIONS, AND PROCEDURES	4
4.0 RESULTS AND DISCUSSION	8
5.0 CONCLUSIONS	12
6.0 REFERENCES	13
APPENDIX A – Publications and Presentations	15
APPENDIX B - Technical annexes	17
B.1 - VENUS semantical model	18
B.2 - Identification of target neural network architectures and activation functions with respect to verification needs	31
B.3 - Definition of Specification Classes	37
B.4 - Methods for Reachability Analysis of ReLU LE-CPSs via MILP	47
B.5 - Low foot-print methods for reachability analysis for feed-forward ReLU LE-CPSs	56
B.6 - Functional evaluation	97
LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS	108

LIST OF FIGURES

Figure 1. NSVerify performance on the TwinStream Dataset.	8
Figure 2. Experimental analysis of NANESVerify for VerticalCAS.	9
Figure 3. Experimental analysis of the VENMAS toolkit on VCAS[2].	9
Figure 4. The VENUS tool architecture.	10
Figure 5. Experimental Analysis of Venus 1.0.	10
Figure 6. Experimental Analysis of Venus 2.0.	10
Figure 7. Input and output of the pose-estimator for the runway detection challenge.	12

1.0 SUMMARY

The overarching objective of the project “Venus: Formal Verification of Neural Systems” concerned the development of the theoretical underpinnings, methods, and tools for the formal analysis of learning-enabled cyber-physical systems (LE-CPS) and their assessment on usecases of industrial relevance within the “Assured Autonomy” project.

Specifically the effort was aimed at:

1. The formalization of the problem of formal verification for open-loop and closed-loop LE-CPS both in terms of decision problem and in terms of specifications to be verified.
2. The development of algorithms and methods for the scalable verification of LE-CPS.
3. The open-source development and release of a toolkit for the methods of point 2.
4. The application of the tool in usecases arising from Technical Area (TA)4 performers.

The effort reached and exceeded all its stated objectives.

In terms of problem formalization, both open-loop and closed-loop systems were formulated mathematically and the complexity of various verification problems was studied. Concerning closed-loop LE-CPS, a wide range of specifications ranging from safety, to temporal and strategic specifications were formalized and integrated into the verification methods. Open-loop systems, such as LE-CPS-based perception systems were formalized in terms of noisy inputs, but also in terms of contrast, luminosity and geometric changes in the input.

The algorithm and methods derived in the effort included both Mixed-integer Linear Programming (MILP)-based methods and abstraction based-methods. The algorithm supported all specifications proposed; during the effort in excess of one order of magnitude scalability gain per phase was achieved, reaching capabilities in excess of 10^7 tuneable model parameters for open-loop systems by the end of the effort.

The Venus toolkit was released at planned intervals to the community and benchmarked both internally and in international competitions, as well as adopted by third parties as tool of choice for the verification for LE-CPS.

A strong and effective working relationship was established with Boeing Research early in the project. This enabled continuous benchmarking of the methods and tools as well as the tailoring of the methods produced to industrially relevant usecases. Several challenge problems were tackled by VENUS during the life of the project. In the final demo VENUS was used to verify the robustness of a visual perception pose-estimator for runway detection consisting of a U-net of over 10^6 rectified linear unit (ReLU) activations. This constitutes, to the best of our knowledge, the most complex neural model used in applications ever verified.

2.0 INTRODUCTION

Background. At time of project solicitation (2017), while Machine Learning (ML) had shown promising new capabilities in a range of society critical tasks, no principled methods existed for providing assurance to ML applications. Vendors routinely used testing as a form of model validation, sometimes paired with simulation, to overcome the difficulty of assuring performance on the intended domain. Both testing and simulation suffered from considerable difficulties, including costs, effort, coverage and actual capabilities. Most importantly, in 2016 it was estimated that testing alone would have been unlikely to ever reach the capability of providing statistical guarantees for complex ML applications such as autonomous vehicles [KP16].

Cyber-physical Systems (CPS) are a class of systems which would particularly benefit from ML capabilities. The resulting ML-based CPS, or Learning-enabled CPS (LE-CPS), may include one or more ML-based components, or Learning-enabled Components (LECs), combined with traditionally designed hardware and software. For LE-CPS to be deployed in safety-critical applications, such as autonomous systems, it is important that their behavior can be validated to be safe and specifications compliant at design time. The overall “assured autonomy” effort responded to the growing need of generating deterministic and quantifiable guarantees and assurance methods for design-time and run-time assurance of LE-CPS.

Within this context, the overarching aim of the VENUS project was to develop pioneering formal methods and tools for the verification of LE-CPSs. To achieve this the following goals were identified:

1. The formalization of LE-CPSs encompassing both traditional computing systems and neural components (LECs) as well as the interactions of with the environment.
2. The development of formal verification techniques for LE-CPSs defined as in 1. These were meant to target both open-loop systems (LECs) consisting of one neural network as well as closed-loop systems (LE-CPSs) interacting with the environment.
 - a. In terms of LECs, the methods were intended to scale from initially targeting shallow LECs of few small layers to deep models of several large layers with hundreds of components. Similarly, feedforward LECs would initially be investigated and in later stages these would be extended to recurrent LECs.
 - b. In terms of LE-CPSs, the methods were intended to consider firstly purely neural systems, governed by feed-forward LECs and later include recurrent LECs, culminating with the analysis of LE-CPSs including both a symbolic and neural components.

Targeted specifications included reachability both for LE-CPSs and LECs, but would be extended to temporal analysis in later phases.

Methods would range from exact to abstraction-based to scale to increasingly large models. A target of one order of magnitude gain in terms of the size of the LECs and LE-CPSs to be analyzed was set for every phase of the project, ranging from 10^2 at program start to 10^5 by the end of the project.

3. The verification methods of point 2 above were to be implemented in VENUS, a novel verification toolkit for LE-CPSs to be released as open-source to the community, which would be capable of verifying systems with thousands of neurons.
4. The methods and tools would be validated on usecases arising from TA4 performers.

Distribution (A) Approved for public release; distribution is unlimited.

Results. The program of work achieved and surpassed all stated objectives.

Specifically,

1. In terms of formalizations, semantics for LECs and LE-CPSs amenable to verification were designed. These ranged from purely neural systems, to neural-symbolic systems culminating in the concept of neuro-symbolic multi-agent systems, encoding LE-CPSs.
2. In terms of verification methods several methods were developed, including:
 - a. Formal verification methods for both LECs and LE-CPSs based on Mixed-integer linear Programming (MILP) were refined from previous work and extended with a novel concept of dependency analysis enabling greater scalability.
 - b. Abstraction-based verification methods for LECs were put forward. These consisted in extensions of methods based on symbolic interval propagation with linear approximations.
 - c. Further abstraction-based methods based on tighter approximations provided by novel Semi-definite Programming (SDP) were developed.

The methods developed targeted concepts of reachability and noise robustness analysis for LECs. These were further extended to support wide classes of robustness perturbations, including contrast and luminosity changes, and generalized to support expressive first-order specifications.

The worst-case computational complexity for these methods was derived and decidable fragments obtained and studied.

3. The VENUS toolkit was released at regular intervals. At each interaction wider classes of models and activation functions were supported. VENUS participated in a Neural Networks (NN) competition scoring among the most performing tools. Experimental VENUS branches were also used in ARCH-COMP, a benchmarking competition for closed-loop systems verifiers. In the final experiments executed on TA4 challenges the tool verified properties for a U-Net model consisting of 2 millions neurons on non-specialized lab workstations.
4. An active collaboration was established early in the effort between Boeing Research and Imperial College London whereby challenge problems posed by Boeing would be solved by suitably extended versions of the VENUS toolkit. The challenge problem of distance estimator for runway taxiing was verified in Phase I. Several challenge problems including a land/go-around instrument-based LEC and a perception based object detector were analyzed in Phase II. Finally, in Phase III a large perception-based runway detector estimating key points was verified and several potential weaknesses identified against several perturbations of interest.

Overall the effort resulted in the publication of over 16 papers in top conferences and journals and well over 20 talks in various forms. The effort contributed to training 5 post-doctoral researchers, most of whom have now moved to tenure track positions.

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

This section summarizes the key methods developed in the effort. The interested reader is referred to the various appendices for further technical details. For ease of understanding, the rest of this section is organized in terms of the target system architecture to be analyzed and reports the progression and tech capabilities reached for the methods derived. For each of these, formal definitions and algorithms are reported together; experimental results are discussed in the next section.

Closed-loop LE-CPSs. This class of systems represents CPS consisting of one or more agents equipped with learning-enabled components and interacting with an environment. At the start of these investigations no method existed to analyze this class of systems.

The first result we obtained is summarized in [ALMP18], where we developed the first formal model for LEC-based agent-environment systems, where the agents are implemented via feed-forward ReLU neural networks. We provided a semantics for these systems by extending the concept of interpreted systems and developed a novel algorithm for their verification. Specifically, we focused on determining whether unwanted states (representing a system violation) are potentially reached by the system during its evolution. We studied several reachability problems for the system, ranging from one-step reachability, to fixed multi-step and arbitrary-step to study the system evolution. We also investigated the decision problem concerning whether an agent, realized via feed-forward ReLU networks may perform an action of interest during a system run. We established NP-Completeness results for the decision problems above and we implemented a practical verification procedure based on MILP to solve the resulting verification problem.

This result was extended in [ABKL20a], where multiple agents, governed by feed-forward ReLU neural networks, interact with a partially-observable, non-deterministic environment. Rather than pure reachability properties we were here concerned with branching time specifications expressed by statements in Computational-Tree-Logic (CTL). In this investigation we showed that verifying these systems against such specifications is undecidable in general. To alleviate this difficulty, we introduced a bounded fragment of CTL for which the verification problem is in coNExpTime and PSPACE-Hard . We showed that this fragment is useful to identify shallow and time-bounded errors in the system. We also introduced a further MILP-based verification algorithm, which we also parallelized for efficiency, for the verification of this class of systems.

These results were combined and extended in [ABKL22], where further experimental results and an optimized implementation was put forward (see next section).

In the approaches discussed above, agents are entirely governed by feed-forward neural networks. However, it is envisaged that LE-CPSs may also be realized through hybrid architectures. In [ABKL20b] we put forward the novel concept of neuro-symbolic multi-agent systems. These are systems in which each agent is endowed with both an LEC, representing the perception function, and a symbolic component, representing the action selection mechanism. While temporal logic is a widely used formalism for capturing specifications, in the past 10 years other, more expressive languages have been put forward to represent AI-based properties of interest. In particular, primitives expressing cooperation are now widely used to capture what teams of agents can achieve collectively during their execution, potentially against a team of adversaries. We adopted Alternating-time Temporal Logic (ATL) in [ABKL20b] as a specification language for neuro-symbolic agents and investigated the verification problem for

Distribution (A) Approved for public release; distribution is unlimited.

these systems. As above, the problem was shown to be undecidable. To mitigate this aspect, a bounded fragment of ATL was adopted. This led us to derive a PSPACE lower bound and a coNEXPTIME upper bound for the verification problem. A novel MILP-based algorithm was devised to verify neuro-symbolic multi-agent systems and a prototype tool, discussed in the next section, was experimentally evaluated.

The results above involve LE-CPSs whose LECs are governed by feed-forward neural networks. While these constitute the large majority of models in applications, recurrent models are also used whenever the model output depends on a sequence of inputs and not just instantaneous ones. To extend our results, this aspect was also addressed.

Firstly, in [AKLP19], we introduced a model for LE-CPSs where the agent is realized by a stateful, ReLU-based recurrent neural network. We defined and addressed their verification by providing an equivalence between verification on bounded traces for recurrent LE-CPSs and feed-forward LE-CPSs on equivalent traces. This led us to a sound and complete verification procedure against specifications expressed in a bounded version of linear-temporal logic (LTL). As discussed later in this document, a drawback of the approach in [AKLP19] is scalability. To mitigate this, we revised the approach in [HL23a] where a new recursive, MILP-based approach for verifying recurrent LE-CPSs was defined. The novel approach uses less than a third of the variables used in the previous one thereby improving its scalability. To the best of our knowledge this constitutes the most scalable approach to verifying recurrent LE-CPS at present.

Open-loop LE-CPSs. The approaches developed for the verification of open-loop LE-CPS focused on targeting LECs comprising a single feedforward ReLU neural network. More complex systems, such as LECs constituted by symbolic and neural layers, are expressible as fragments of the closed-loop approaches above. Differently from closed-loop systems, where expressivity of the system and the specification language is an important issue, the primary target in this line is the scalability of the resulting approaches.

The initial approach we undertook in this line was, in common with [ALMP18], an optimization based approach built on MILP whereby an LEC is analyzed either for reachability or robustness. In this line of research, robustness is defined as the property stating that the system’s output does not change (or changes minimally) in the neighborhood of an input. An often analyzed robustness specification concerns noise robustness as measured by epsilon balls on an input. A model is said to be noise robust for that input if its output does not change in the dense noise neighborhood defined by the epsilon ball under consideration. Following on from our initial MILP approach the concept of dependency analysis was developed and used for the rest of the effort. Specifically, in [BKKLM20] an automated procedure that exploits dependency relations between the ReLU nodes was developed. This allows an improvement on the underlying branch-and-bound verification procedure, by pruning the search tree in accordance with the dependency relation. This algorithm was augmented with methods for input domain splitting and symbolic interval propagation. This approach was implemented in the first release of VENUS, the verification toolkit that was used throughout the effort to address challenge problems from TA4 performers.

Dependency-based verification was further enhanced in [KL21] whereby dependency relations are also used for branching on the ReLU states. This enabled the realization of a further version of VENUS whereby the resulting MILP formulations require much fewer integrality constraints.

Distribution (A) Approved for public release; distribution is unlimited.

The MILP-based approaches of [BKKLM20, KL21] are exact and complete, in the sense that, barring computational constraints, they will always return the solution to the verification problem. During the effort and in line with the objectives, we also pursued abstraction-based approaches. In particular, we developed a method of symbolic interval propagation (SIP), where the computation of the LEC is approximated symbolically, potentially sacrificing completeness while retaining soundness. Our first result in this line is [HL20] where LECs based on ReLU, but also Sigmoid and Tanh activation functions are verified for robustness. The method extended plain SIP by using gradient-descent to locate counter-examples from spurious solutions generated by the associated LP problems. The approach also included a novel adaptive splitting strategy intended to refine the nodes with the greatest impact on the output of the network.

In SIP-based approaches the ReLU function is relaxed linearly by considering an overapproximation. In the literature horizontal and triangle relaxations are widely used, leading to varying results. A further advancement on SIP was achieved in [HKL21] where the choice of the ReLU relaxation to be used at each node is determined via optimization approach. A combination of this approach with dependency-based MILP and further optimizations constituted the basis for the open-source VENUS2 release which was used in the challenges of Phase II. In all the results above specifications varied from noise robustness to contrast robustness and luminosity robustness, thereby enabling a more comprehensive analysis of the performance of the LEC under investigation.

SIP-based approaches such as the ones developed in this effort remain at time of writing the most computationally advantageous approaches for verifying LECs. However, they are intrinsically limited by the approximation that is carried out when linearizing the ReLU activations. The compounded effect of these approximations, i.e., their compound looseness, creates a barrier to verifiability. In another line of work we developed approaches in which this approximation is tighter, because it is based on more precise representations of the ReLU function. More specifically, in [BKLZ21] we introduced an efficient and tight layer-based semidefinite relaxation for verifying LECs. The approach is demonstrably more efficient than previous SDP approaches as well as being tighter.

The approach was further improved in [LZL22] by adding to the formulation linear cuts constructed by using the reformulation-linearisation technique (RLT), thereby resulting in a layer RLT-SDP relaxation considerably tighter than previous approaches. This approach has been further improved [LZL23, LZB23] after this project ended, resulting in tighter approximations still.

Additional Methods. Additional methods were pursued in the life of this effort that were not originally envisaged. These extend and naturally arise from the results of the project.

Firstly, while efforts on verification methods resulted in gains of one order of magnitude for each of the effort’s phases, the resulting methods are still not capable of addressing the ever increasing complexity of the most challenging applications, e.g., perception systems used in autonomous vehicles. An intrinsic difficulty of all verification methods for LE-CPSs stems from the use of ReLU functions as neuron activations. Yet, ReLU are not the only functions that enable efficient learning on neural models. In this context, [HLL23] showed that an alternative class of functions, called Parametric ReLU activation, offer the benefits of plain ReLU functions but result in verification problems that are up to one order of magnitude easier to solve than those based on

Distribution (A) Approved for public release; distribution is unlimited.

standard ReLUs. This opens up the way of further gains in assurance by adopting slightly different training functions at no precision loss.

Secondly, the experimental evaluation conducted in the program (see next section) evidenced that LE-CPSs suffer from potential vulnerabilities even when trained with state-of-the-art methods. A future challenge for developers not only concerns verification, but also how to remove fragilities from the model without lowering its performance on the test set. To investigate these issues an algorithmic concept of model repair, closely connected to verification, was investigated [HLL22]. The method assumes no access to the training set. The algorithm, based on gradient-search, performs localized modifications to the network's weights such that misclassifications are repaired while minimally affecting network accuracy on correctly classified inputs. The theoretical properties, including its soundness and minimality were also shown. Work on model repair was generalized to robust learning in [HL23b] which is presently the object of further investigations.

Lastly, during the effort we observed that the methods and tools developed during this program could potentially be used in other nearby areas. One example of this is explainable AI where explanations are required to be robust. To illustrate the technology and facilitate cross-fertilization in [LL23] the problem of generating contrastive explanations with formal robustness guarantees was investigated. Two novel verification-based algorithms were introduced to identify non-robust explanations generated by other methods and to generate contrastive explanations augmented with provable robustness certificates. A feature of this approach is that robust explainability can be established by the verification methods and tools developed elsewhere in this effort.

4.0 RESULTS AND DISCUSSION

Summary. The results obtained exceeded the results anticipated in the proposal. As also discussed at page 3 of this report, the key achievements include:

1. The formalization of LECs, LE-CPSs and definition of their verification problem for design-time assurance.
2. The development of exact and approximated algorithms, ranging from MILP-based, to SIP-based and SDP-based for their verification.
3. The formulation of specialized optimizations for said algorithms achieving 1 order of magnitude gain per phase culminating in methods tackling the verification of local robustness for LECs in the millions of nodes.
4. The production and open-source release of the VENUS toolkit, and other related checkers, thereby empowering the community with state-of-the-art verifiers.
5. The use of VENUS in challenges formulated by Boeing of increased level of difficulty and tech-readiness ranging from a simulated centerline distance estimator for runway taxiing to a detector system for key-points estimation for runway detectof over 1M ReLU nodes.

In the following we summarize the key results achieved in the various lines but refer to the technical annexes for details.

Closed-loop LE-CPSs. Throughout the effort the verification methods for LE-CPSs were implemented in a toolkit called NSVERIFY. Initially NSVERIFY solved single-step, multi-step state reachability decision problems and multi-step action executability problems. NSVERIFY takes as input a neural agent and a neurally-approximated environment. Upon invocation, the tool builds the set of the corresponding linear constraints which are then solved by the MILP back-end. A result presented to the user, either SAT, indicating that an output state is reachable (or, an output action executable) from the input states or an UNSAT result representing

Tool	Success rate	Average runtime	Number of wins
NSVERIFY	62.96%	2106.8s	16
RELUPLEX	53.08%	2135.5s	4
MIP	69.14%	2430.2s	7
BAB	29.63%	5212.5s	5
PLANET	64.20%	2807.9s	29

Figure 1. NSVerify performance on the TwinStream Dataset from [ALMP18].

unreachability of the output states (or, impossibility of the actions). In case of SAT, a trace showing the states and actions corresponding to the solution are also given.

NSVERIFY was evaluated on a number of in-house benchmarks including inverted pendulum challenges from OpenAIGym. NSVERIFY achieved state-of-the-art performance when released performing more efficiently than other tools on the Twin Stream Data Set (see Figure 1). NSVERIFY was also used as a backend for the verification of RNN-based LC-CPSs [AKLP19],

where the verification problem for RNN-based agents was recast as verification of feed-forward LE-CPS via unrolling as input-on-start and input-on-demand [APLK19].

NSVERIFY evolved into the NANESVerify toolkit, which supports the verification of on-deterministic environments [ABKL20]. This enabled the verification of an instance of VerticalCAS[2], a prototype advisory system for 2 aircrafts generating vertical advisories to the ownship pilot to avoid near mid-air collisions.

k	COMP-PAR				COMP-SEQ						MONOLITHIC					
	-19.5	-22.5	-25.5	-28.5	-19.5	-22.5	-25.5	-28.5	C	V	-19.5	-22.5	-25.5	-28.5	C	V
1	0.629s	0.608s	0.649s	0.652s	0.728s	0.819s	0.737s	0.750s	3037	2993	0.039s	0.039s	0.041s	0.042s	3190	2996
2	2.901s	2.730s	1.092s	1.429s	5.392s	5.594s	0.623s	0.618s	6668	5981	4.399s	6.450s	1.444s	3.323s	6974	5987
3	10.67s	1.716s	1.918s	1.824s	26.06s	0.986s	0.961s	0.964s	10300	8969	23.33s	14.58s	12.79s	13.59s	10759	8978
4	39.58s	40.91s	2.474s	2.570s	109.9s	108.7s	1.404s	1.417s	13930	11957	-	-	377.5s	29.96s	14542	11969
5	145.6s	156.3s	159.8s	3.830s	433.5s	481.2s	512.4s	2.244s	17560	14945	-	-	-	751.1s	18325	14960
6	797.4s	544.8s	573.5s	568.8s	2174.s	1639.s	1826.s	1859.s	21198	17933	-	-	-	-	22116	17951

Figure 2. Experimental analysis of NANESVerify for VerticalCAS, where k indicates the number of steps in the system evolution and the different tables represent the particular solving method (Composition, or Monolithic).

Various streams and experimental versions of the toolkits above were combined and extended into VENMAS [ABKL20] targeting the verification of LE-CPS against specifications expressing strategic interplay for the agent in the system. VENMAS constitutes the first available tool for the verification of neuro-symbolic LE-CPS modeled in a variant of interpreted systems semantics combining both neural and symbolic systems layers. While results were novel and promising, it also became clear that scalability of the methods available at the time was a challenge. Due to this particular efforts were devoted to conquer scalability in the neural layer, as discussed below.

h_I	COMP-PAR				COMP-SEQ				MONOLITHIC			
	-30	10	50	90	-30	10	50	90	-30	10	50	90
φ_1^1	0.33	0.39	0.34	0.48	0.32	0.34	0.32	0.97	0.04	0.08	0.05	0.04
φ_2^1	1.67	4.07	13.63	13.88	1.29	3.16	41.82	42.64	0.53	4.17	0.16	0.16
φ_3^1	104.29	-	-	-	59.48	-	-	-	303.35	994.45	0.67	0.85
φ_4^1	-	-	-	-	-	-	-	-	-	-	3.20	3.40
φ_5^1	-	-	-	-	-	-	-	-	-	-	9.63	9.22
φ_1^2	0.35	0.37	0.35	0.37	0.32	0.37	0.32	0.57	0.04	0.08	0.05	0.04
φ_2^2	1.23	3.07	2.21	2.47	1.17	2.99	5.66	5.76	0.52	4.25	0.16	0.16
φ_3^2	55.20	564.85	69.38	100.93	22.14	1823.65	213.81	309.56	906.68	933.99	0.66	0.84
φ_4^2	-	-	-	-	-	-	-	-	-	-	3.21	3.38
φ_5^2	-	-	-	-	-	-	-	-	-	-	9.51	9.23

Figure 3. Experimental analysis of the VENMAS toolkit on VCAS[2] from [ABKL20]. In the table the columns represent different encodings to the underlying solver and the various specifications encode strategic interplay in the agents in the system.

Open-loop LE-CPSs. The methods described above were implemented in VENUS, an open-source verification toolkit for neural networks [BKKLM20, KL21]. VENUS uses the big-M encoding for solving LE-CPS verification problems, and tightens the corresponding linear relaxations by adding dependency cuts and ideal cuts to the MILP programs. The cuts are added at runtime through solver callbacks (see Figure 4 for the tool architecture). The key advancement

of the VENUS toolkit consists in its dependency analysis procedure augmented by symbolic interval propagation and input-domain splitting.

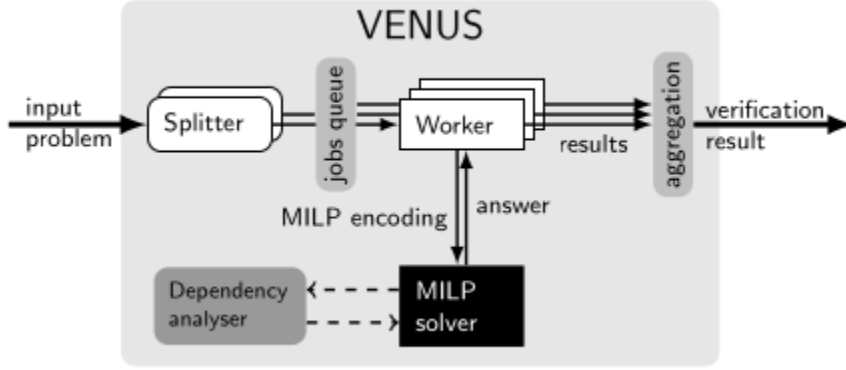


Figure 4. The VENUS tool architecture.

By means of dependency analysis Version 1 of the tool obtained state-of-the-art performance when it was released at the end of Phase I (see Figure 5).

	MNIST (100 verification queries)						CIFAR-10 (100 verification queries)						ACAS XU (172 verification queries)					
	n_s	t_{all}	t_{solved}	av_s	$\frac{t_{all}}{t_{Venus}}$	$\frac{t_{all}}{t_{all}}$	n_s	t_{all}	t_{solved}	av_s	$\frac{t_{all}}{t_{Venus}}$	$\frac{t_{all}}{t_{all}}$	n_s	t_{all}	t_{solved}	av_s	$\frac{t_{all}}{t_{Venus}}$	$\frac{t_{all}}{t_{all}}$
Venus	100	5,953.46	573.38	9.10	—	—	100	857.11	560.04	7.36	—	—	170	19,642.57	5,527.76	36.36	—	—
Marabou	0	86,400.00	—	—	14.51	—	0	86,400.00	—	—	100.80	—	156	140,916.96	75,747.78	498.3	7.17	—
Neurify	65	126,007.24	7.00	0.11	21.17	—	76	87,178.46	778.46	10.24	101.71	—	167	23,628.75	2,555.38	16.81	1.2	—
NSVerify	95	26,906.81	2,515.15	39.9	4.52	—	100	6,898.64	3,460.41	45.53	8.04	—	6	86,400.00	—	—	30.77	—

Figure 5. Experimental Analysis of Venus 1.0 against commonly used benchmarks from [BKKLM20].

Several releases of VENUS were made during the effort as the tool matured. In a further enhancement [HKL20] a sophisticated heuristic for an efficient choice in the different linear relaxations was added to the software. This enabled the verification of large VGG16 models. In

Model	Radius	Venus2		Eran		Marabou		MIPVerify		Neurify		Nnenum		Verinet		Venus	
		ver	t	ver	t	ver	t	ver	t	ver	t	ver	t	ver	t	ver	t
ACASXU	-	186	11.3	76	1065.0	149	497.9	125	807.1	179	98.9	186	1.8	-	-	184	123.3
MNIST2	0.02	25	0.3	25	0.5	24	151.0	25	0.3	23	283.9	25	1.7	25	50.1	25	0.6
	0.05	25	2.8	24	109.3	7	1430.8	25	2.9	20	395.2	25	19.1	22	216.0	25	5.5
MNIST4	0.02	25	4.3	25	18.6	21	325.3	25	20.9	23	213.7	24	96.5	20	360.1	24	83.6
	0.05	25	383.12	22	1688.0	0	1800.0	6	1407.0	1	1728.0	6	1267.8	15	1080.2	3	1603.1
MNIST6	0.02	25	120.9	18	504.4	7	325.3	19	362.5	18	512.7	21	293.0	16	648.1	9	1112.2
	0.05	25	605.7	0	1800.0	0	1800.0	0	1800.0	1	1728.0	6	1270.9	8	1237.9	3	1595.0
ALL		336	96.6	171	896.1	218	727.6	229	581.5	265	417.3	293	237.0	101	597.2	275	376.0

Figure 6. Experimental Analysis of Venus 2.0 from [KL21].

release 2.0 [KL21] VENUS included further advancements on dependency-based branching and efficiency gains in the performance of algebraic operations enabling the verification of challenge problems of Phase II consisting of LE-CPS in the region of hundreds of thousands of neurons. VENUS was further enhanced towards the end of Phase III thereby enabling the verification of challenge problems in the region of millions of non-linearities. Overall the development of methods and tools for the verification of LE-CPS saw performance increases of over one order of magnitude per year, starting at approximately 10^2 nodes and terminating at well over 10^6 .

As mentioned, other approaches for the verification of LE-CPS were pursued during the effort. Since linear relaxations generate loose relaxations when applied to deep models, tighter relaxations were attempted. In particular the line that originated from [LZL22] and further developed in [LZL23, LZB23] uses SDP-based encoding (a lighter form of quadratic encodings) to obtain relaxations that are tighter than any of the linear approaches here pursued. At present the computational costs of SDP-based approaches are higher than the best performing linear approaches, thereby resulting in overall worse performance, but work continues in this direction.

Solutions to Challenge Problems. The approaches below found application from the start of the effort to challenge problems developed by Boeing Research that were pursued jointly. In Phase I the team used VENUS to verify the robustness of an LEC of approximately 10k nodes measuring the distance from the runway centerline. VENUS identified the thresholds for robustness and identified fragilities for the model. In Phase II VENUS was used to verify the robustness of an LEC for object detection with open categories of approx 200,000 neurons. In the same challenge a decision making LEC on tabular data for land/go-around advisories was also analyzed. The reported results [KKLLM21] identified fragilities and validated the robustness of models up to a previously unknown threshold. Finally, in Phase III, VENUS 2.0 was further optimized and used to verify a pose estimator, implemented as a U-Net of over 2M non-linearities. As above, this enabled a fine grain verification on the robustness of the model to noise, contrast and luminosity perturbations as well as identifying various fragilities [KLECML23]. At time of writing and to our knowledge, this constitutes the largest and most complex LEC ever analyzed via verification tools.

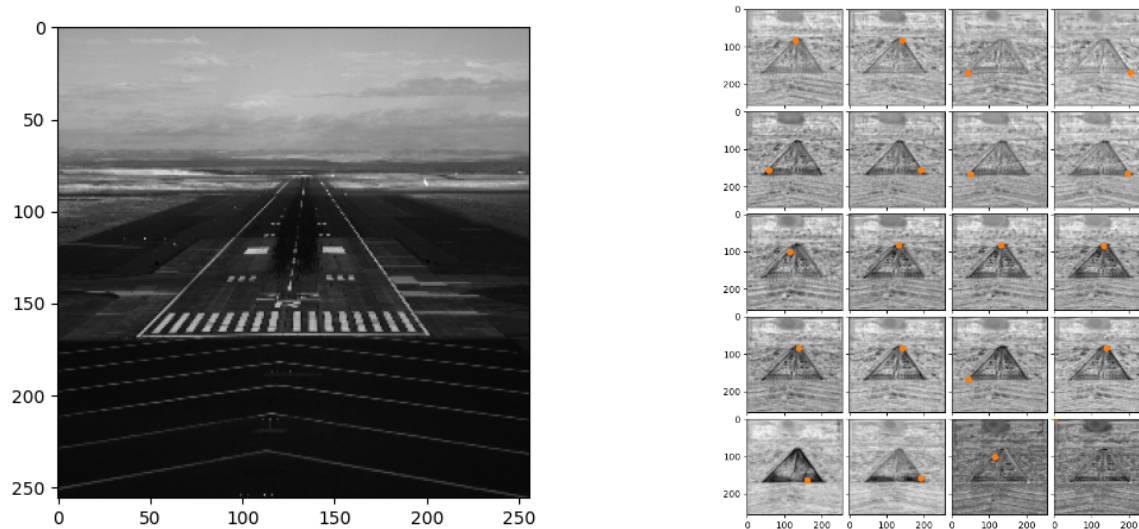


Figure 7. Input and output of the pose-estimator for the runway detection challenge studied in [KLECML23].

5.0 CONCLUSIONS

The project VENUS developed a variety of methods for the formalization of various LE-CPSs, including open-loop and closed-loop systems, as well as purely neural and neurosymbolic.

The effort achieved and surpassed all its stated objectives in terms of formalizations, algorithmic developments, tools, and applications to challenge problems, reaching a capability of verifying LE-CPS in the millions of nodes and tens of millions of tunable parameters.

While the results exceeded expectations, challenges in deploying LE-CPS in production environments remain.

Firstly, scalability remains a challenge. Further work is required to reach the complexity of models presently used in production, particularly in vision challenges, which presently surpass 10^8 neurons.

Secondly, most verification efforts within the Assured Autonomy effort focused on verification against noise perturbations. While this is important, more effort is required to model and solve verification challenges against other perturbations of interest, including, but not limited to, pose changes, and background changes.

Thirdly, the development of verification technology for LE-CPS evidenced the widespread fragility of systems as presently constructed. Efforts are required towards developing methods for automated robustification and repair following the discovery of fragilities via verification tools such as VENUS.

6.0 REFERENCES

- [KLECML23] P. Kouvaros, F. Leofante, B. Edwards, C. Chung, D. Margineantu, A. Lomuscio. Verification of Semantic Key Point Detection for Aircraft Pose Estimation. Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning (KR2023). Rhodes, Greece. AAAI Press. 2023. To appear.
- [HLL23] P. Henriksen, F. Leofante, A. Lomuscio. Verification-friendly Networks: the Case for Parametric ReLUs. Proceedings of the 36th International Conference on Neural Networks (IJCNN23). Gold Coast, Australia. IEEE Press. 2023. To appear.
- [HL23a] M. Hosseini, A. Lomuscio. Bounded and Unbounded Verification of RNN-Based Agents in Non-deterministic Environments (extended abstract).. Proceedings of the 22nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS23). London, UK. IFAAMAS Press. 2023. To appear.
- [LL23] F. Leofante, A. Lomuscio. Towards Robust Contrastive Explanations for Human-Neural Multi-Agent Systems (extended abstract). Proceedings of the 22nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS23). London, UK. IFAAMAS Press. 2023. To appear.
- [HL23b] P. Henriksen, A. Lomuscio. Robust Training of Neural Networks against Bias Field Perturbations.. Proceedings of the 37th AAAI Conference on Artificial Intelligence (AAAI23). Washington, DC, USA.
- [LZL23] J. Lan, Y. Zheng, A. Lomuscio. Iteratively Enhanced Semidefinite Relaxations for Efficient Neural Network Verification. Proceedings of the 37th AAAI Conference on Artificial Intelligence (AAAI23). Washington, DC, USA. 2023.
- [LBL23] J. Lan, B. Brueckner, A. Lomuscio. A Semidefinite Relaxation based Branch-and-Bound Method for Tight Neural Network Verification. Proceedings of the 37th AAAI Conference on Artificial Intelligence (AAAI23). Washington, DC, USA. 2023.
- [LZL22] J. Lan, Y. Zheng, A. Lomuscio. Tight Neural Network Verification via Semidefinite Relaxations and Linear Reformulations. Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI22). Virtual Conference. 2022.
- [ABKL22] M. Akintunde, E. Botoeva, P. Kouvaros, A. Lomuscio. Formal Verification of Neural Agents in Non-deterministic Environments. Journal of Autonomous Agents and Multi-Agent Systems, Vol 36(1). Springer. 2022.
- [HKL21] V. Hashemi, P. Kouvaros, A. Lomuscio. OSIP: Tightened Bound Propagation for the Verification of ReLU Neural Networks. Proceedings of the 19th International Conference on Software Engineering and Formal Methods (SEFM21). Virtual Conference. 2021.

- [KKLLM21] P. Kouvaros, T. Kyono, F. Leofante, A. Lomuscio, D. Margineantu, D. Osipychiev, Y. Zheng. Formal Analysis of Neural Network-based Systems in the Aircraft Domain. Proceedings of the 24th International Symposium on Formal Methods (FM21). Virtual conference. 2021.
- [KL21] P. Kouvaros, A. Lomuscio. Towards Scalable Complete Verification of ReLU Neural Networks via Dependency-based Branching. Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI21). Virtual conference. Ijcai.org. 2021.
- [BKLZ21] B. Batten, P. Kouvaros, A. Lomuscio, Y. Zheng. Efficient Neural Network Verification via Layer-based Semidefinite Relaxations and Linear Cuts. Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI21). Virtual conference. Ijcai.org. 2021.
- [HL20] P. Henriksen, A. Lomuscio. Efficient Neural Network Verification via Adaptive Refinement and Adversarial Search. Proceedings of the 24th European Conference on Artificial Intelligence (ECAI20). Santiago de Compostela, Spain. 2020.
- [ABKL20b] M. Akintunde, E. Botoeva, P. Kouvaros, A. Lomuscio. Verifying Strategic Abilities of Neural-symbolic Multi-agent Systems. Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning (KR20). Virtual Conference. IJCAI Press. 2020.
- [ABKL20a] M. Akintunde, E. Botoeva, P. Kouvaros, A. Lomuscio. Formal Verification of Neural Agents in non-Deterministic Environments. Proceedings of the 19th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS20). Virtual Conference. IFAAMAS Press. 2020.
- [BKKLM20] E. Botoeva, P. Kouvaros, J. Kronqvist, A. Lomuscio, R. Misener. Efficient Verification of Neural Networks via Dependency Analysis. Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI20). New York City, New York. AAAI Press. 2020.
- [AKLP19] M. Akintunde, A. Kevorchian, A. Lomuscio, E. Pirovano. Verification of RNN-Based Neural Agent-Environment Systems. Proceedings of the 33th AAAI Conference on Artificial Intelligence (AAAI19). Honolulu, HI, USA. AAAI Press. 2019.
- [ALMP18] M. Akintunde, A. Lomuscio, L. Maganti, E. Pirovano. Reachability Analysis for Neural Agent-Environment Systems. Proceedings of the 16th International Conference on Principles of Knowledge Representation and Reasoning (KR18). Tempe, Arizona. AAAI Press. 2018.
- [KP16] N. Kalra, S. Paddock. Driving to Safety - How Many Miles of Driving Would it Take to Demonstrate Autonomous Vehicle Reliability? RAND Corporation, 2016. Cited IN BAA HR001117S0045. DARPA, 2017.

APPENDIX A – PUBLICATIONS AND PRESENTATIONS

List the dates, times, title, event and speakers of any presentations made under this effort and the title author and publication information for any publication made under this effort.

Publications.

- P. Kouvaros, F. Leofante, B. Edwards, C. Chung, D. Margineantu, A. Lomuscio. Verification of Semantic Key Point Detection for Aircraft Pose Estimation. Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning (KR2023). Rhodes, Greece. AAAI Press. 2023. To appear.
- P. Henriksen, F. Leofante, A. Lomuscio. Verification-friendly Networks: the Case for Parametric ReLUs. Proceedings of the 36th International Conference on Neural Networks (IJCNN23). Gold Coast, Australia. IEEE Press. 2023. To appear.
- M. Hosseini, A. Lomuscio. Bounded and Unbounded Verification of RNN-Based Agents in Non-deterministic Environments (extended abstract). Proceedings of the 22nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS23). London, UK. IFAAMAS Press. 2023. To appear.
- F. Leofante, A. Lomuscio. Towards Robust Contrastive Explanations for Human-Neural Multi-Agent Systems (extended abstract). Proceedings of the 22nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS23). London, UK. IFAAMAS Press. 2023. To appear.
- J. Lan, Y. Zheng, A. Lomuscio. Tight Neural Network Verification via Semidefinite Relaxations and Linear Reformulations. Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI22). Virtual Conference. 2022.
- M. Akintunde, E. Botoeva, P. Kouvaros, A. Lomuscio. Formal Verification of Neural Agents in Non-deterministic Environments. Journal of Autonomous Agents and Multi-Agent Systems, Vol 36(1). Springer. 2022.
- V. Hashemi, P. Kouvaros, A. Lomuscio. OSIP: Tightened Bound Propagation for the Verification of ReLU Neural Networks. Proceedings of the 19th International Conference on Software Engineering and Formal Methods (SEFM21). Virtual Conference. 2021.
- P. Kouvaros, T. Kyono, F. Leofante, A. Lomuscio, D. Margineantu, D. Osipychiev, Y. Zheng. Formal Analysis of Neural Network-based Systems in the Aircraft Domain. Proceedings of the 24th International Symposium on Formal Methods (FM21). Virtual conference. 2021.

P. Kouvaros, A. Lomuscio. Towards Scalable Complete Verification of ReLU Neural Networks via Dependency-based Branching. Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI21). Virtual conference. Ijcai.org. 2021.

B. Batten, P. Kouvaros, A. Lomuscio, Y. Zheng. Efficient Neural Network Verification via Layer-based Semidefinite Relaxations and Linear Cuts. Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI21). Virtual conference. Ijcai.org. 2021.

M. Akintunde, E. Botoeva, P. Kouvaros, A. Lomuscio. Verifying Strategic Abilities of Neural-symbolic Multi-agent Systems. Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning (KR20). Virtual Conference. IJCAI Press. 2020.

M. Akintunde, E. Botoeva, P. Kouvaros, A. Lomuscio. Formal Verification of Neural Agents in non-Deterministic Environments. Proceedings of the 19th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS20). Virtual Conference. IFAAMAS Press. 2020.

E. Botoeva, P. Kouvaros, J. Kronqvist, A. Lomuscio, R. Misener. Efficient Verification of Neural Networks via Dependency Analysis. Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI20). New York City, New York. AAAI Press. 2020.

M. Akintunde, A. Kevorchian, A. Lomuscio, E. Pirovano. Verification of RNN-Based Neural Agent-Environment Systems. Proceedings of the 33th AAAI Conference on Artificial Intelligence (AAAI19). Honolulu, HI, USA. AAAI Press. 2019.

M. Akintunde, A. Lomuscio, L. Maganti, E. Pirovano. Reachability Analysis for Neural Agent-Environment Systems. Proceedings of the 16th International Conference on Principles of Knowledge Representation and Reasoning (KR18). Tempe, Arizona. AAAI Press. 2018.

APPENDIX B - TECHNICAL ANNEX

This appendix reports technical details of the approaches previously discussed.

VENUS semantical model

Elena Botoeva

Panagiotis Kouvaros

Alessio Lomuscio

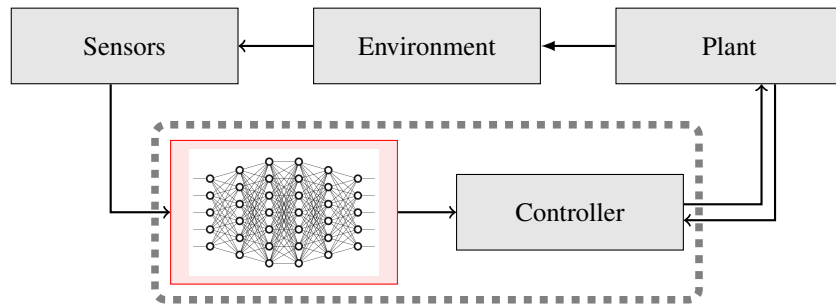
1 Introduction

This report puts forward a formal model for Cyber-Physical Systems (CPSs) equipped with Learning-Enabled Components (LECs) that are implemented as Neural Networks (NNs). Formal representations for NNs and CPSs have separately been studied within a rich body of research work. However there is presently little formal treatment that allows for their integration. Whereas formalisms for NNs consider an NN in isolation, models for CPSs do not account for Learning-Enabled Components (LECs) whose output is governed by NNs. This is very problematic when wishing to reason about the behaviours of CPSs equipped with NNs, henceforth Learning-Enabled CPSs (LE-CPSs). In LE-CPSs a LEC is not operating in isolation but participates in complex patterns of interactions between different components and the environment where these are situated.

Figure 1 depicts the architecture of LE-CPSs. In most application domains LECs act as either perception modules or controllers that take input from the environment and pass their output to a plant. For some applications, e.g. image classification, the environment is fully observable, i.e., the LECs take as input all information necessary to make a decision, whereas in others, e.g. autonomous cars, the environment is only partially observable, i.e., the LECs cannot make an optimal decision on the information available.

For instance steering systems for autonomous cars and airplanes take as input sensory data that are passed to a LEC. The LEC decides the state of the environment, e.g. the cross track error of the runway, where the vehicle is navigating. The output of the LEC is then passed to a (possibly learning-enabled) controller that controls the steering angle of the vehicle. The vehicle

Figure 1: The architecture of LE-CPSs



Distribution (A) Approved for public release; distribution is unlimited.

itself participates in a larger closed-loop system comprising not only the environment but also several other vehicles. So any control decision cannot simply rely on sensory data but needs to account for the interactions with the rest of the participants in the system.

In view of the above this deliverable aims at formalising closed-loop systems composed of a finite number of LECS. The key criteria driving the derivation of the model is its amenability to *complete verification techniques*, i.e., methods whose output can always be used to draw conclusions on the behaviour of the system being modelled, and its expressive power to represent the TA4 use-cases. To enable the construction of complete verification techniques, the semantics targets the modelling of LE-CPSs whose environments can either be approximated by sets of linear equations or represented by NNs, as opposed to the standard modelling practices where the environments are represented as continuously evolving variables. The class of LE-CPSs generated under these constraints is both amenable to exact verification (by building exact encodings into mixed integer programming and SMT programs) and can be used to reason about the TA4 use cases. To account for partially observable environments the semantics allow for temporal evolutions where every next state of the environment is non-deterministically determined (among a set of possible next-states), and to enable the modelling of fully observable environments the temporal evolution of the environment can be given in a deterministic fashion.

The rest of the deliverable is structured as follows. In Section 2 we provide basic definitions of two classes of neural networks, feed-forward and recurrent, that are relevant to this report. Then, in Section 3 we define a model for LE-CPSs, and in Section 4 present several concrete instances of that model.

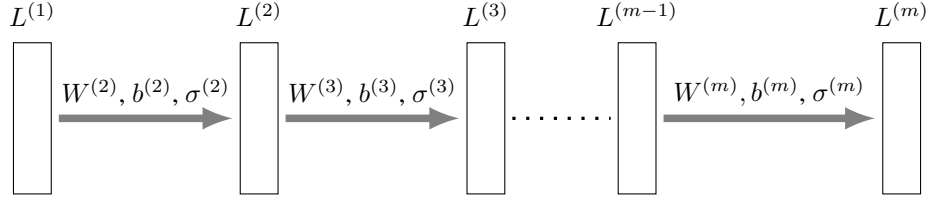
2 Preliminaries

In this section we present concepts helpful in understanding the rest of the report. In particular, we introduce the classes of neural networks that we consider. For more details on neural networks we refer to [5].

2.1 Feed-forward neural networks.

Feed-forward neural networks are one of the simplest classes of neural networks that consist of multiple *hidden layers* and admit no cycles [12]. A *feed-forward neural network (FFNN)* N is made up of m layers, $m \geq 2$, each of which consists of multiple *nodes* that are single computation units combining input from the values of nodes in the previous layer and producing an output to be used in successive layers. We denote the i th layer of N by $L^{(i)}$, the number of nodes in $L^{(i)}$ by p_i , and refer to $L^{(1)}$ as the *input layer*, to $L^{(m)}$ as the *output layer*, and to all other layers $L^{(i)}$, $2 \leq i < m$, as *hidden layers* of the network. Each layer $L^{(i)}$, $i \geq 2$, is associated *weight matrix* $W^{(i)} \in \mathbb{R}^{p_i \times p_{i-1}}$, a *bias vector* $b^{(i)} \in \mathbb{R}^{p_i}$, and an *activation function* $\sigma^{(i)}$. The latter, for each node, applies a transformation to a linear combination of the *values* of the input nodes, thus defining the *value* of the node, which can then be passed into subsequent nodes. We can depict the layers, their parameters and connections as follows:

Distribution (A) Approved for public release; distribution is unlimited.



Neural networks are generally used to learn highly non-linear, non-programmatic functions; upon training, the network computes an approximation of such a function by means of the weights and biases learned. Next, we formally define the function computed by a network.

Definition 2.1 (Function computed by FFNN). *Let N be an FFNN. We define the computed function for $L^{(i)}$, $i \geq 2$, denoted $f_N^{(i)} : \mathbb{R}^{p_{i-1}} \rightarrow \mathbb{R}^{p_i}$, by $f_N^{(i)}(\mathbf{x}) = \sigma^{(i)}(W^{(i)}\mathbf{x} + b^{(i)})$.*

For an m -layer FFNN N , the computed function $f_N : \mathbb{R}^{p_1} \rightarrow \mathbb{R}^{p_m}$ for N , is defined as $f_N(\mathbf{x}) = f^{(m)}(f^{(m-1)}(\dots(f^{(2)}(\mathbf{x}))))$.

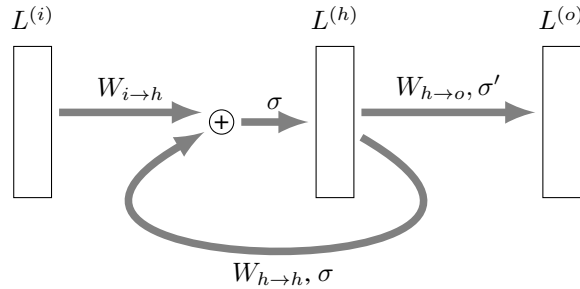
In this report we mention networks utilising *Rectified Linear Unit (ReLU)* activation functions, defined as $\text{ReLU}(x) = \max(0, x)$, where x represents the linear combination of values from incoming nodes in the previous layer. In the following we will refer to feed-forward neural networks which exclusively use ReLU activation functions throughout all hidden layers as *ReLU-FFNNs*.

2.2 Recurrent neural networks.

Recurrent neural networks (RNNs) are a class of neural networks with cyclic connections between the nodes. Differently from FFNNs, whose output is always the same given a certain single input, RNNs are designed to process sequences of data, and are equipped with a state that evolves over time. As a result, the behaviour of an RNN depends on the history of their inputs. We formally define RNNs below.

A single-layer *recurrent neural network* (RNN) R with h hidden units and input size i and output size o is associated with the weight matrices $W_{i \rightarrow h} \in \mathbb{R}^{i \times h}$, $W_{h \rightarrow h} \in \mathbb{R}^{h \times h}$ and $W_{h \rightarrow o} \in \mathbb{R}^{h \times o}$, and the two activation functions $\sigma : \mathbb{R}^h \rightarrow \mathbb{R}^h$ and $\sigma' : \mathbb{R}^o \rightarrow \mathbb{R}^o$. The weight matrix $W_{i \rightarrow h}$ defines the weights of the connections between input units and hidden units, $W_{h \rightarrow h}$ defines the weights of the connections between hidden units, and $W_{h \rightarrow o}$ defines the weights of the connections between hidden units and output units. We refer to the value of the hidden units as the *state* of the network, due to the self-loop introduced by $W_{h \rightarrow h}$. The activation function σ is applied to the state before it is passed onto the next time step, and σ' is applied to the state of the network at a given time step, in order to compute the output of the network at that time step.

We can schematically depict the structure and the connections of an RNN as follows.



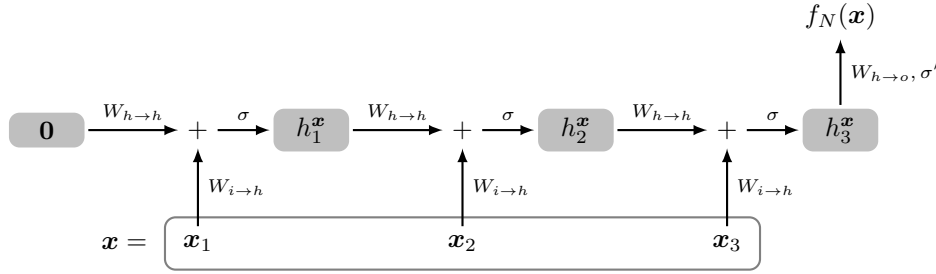
Distribution (A) Approved for public release; distribution is unlimited.

Definition 2.2 (Function computed by RNN). Let N be an RNN with weight matrices $W_{i \rightarrow h}$, $W_{h \rightarrow h}$ and $W_{h \rightarrow o}$. Let $\mathbf{x} \in (\mathbb{R}^i)^n$ denote an input sequence of length n where each element of the sequence is a vector of size i , with $\mathbf{x}_t$ denoting the t -th vector of $\mathbf{x}$. We define $h_0^x = \mathbf{0}$ as a vector of 0s. For each time step $1 \leq t \leq n$, we define:

$$h_t^x = \sigma(W_{h \rightarrow h}h_{t-1}^x + W_{i \rightarrow h}\mathbf{x}_t).$$

Then, the output of the RNN is given by $f_N(\mathbf{x}) = \sigma'(W_{h \rightarrow o}h_n^x)$.

We can visualise the computation of an RNN for a sequence of length 3 as follows:



In the following we denote by *ReLU-RNNs* the recurrent networks where the activation function σ is ReLU.

The ability of RNNs to recall information from previous time steps has traditionally made them useful in language-related tasks such as text prediction [8]. Importantly, they are also used as the basis of controllers for dynamic systems [6, 10, 7]. Therefore, since RNNs define a stateful component, they can be taken as a basis for a stateful learning-enabled component as we do in Section 4.3.

3 Formal Framework for Verification of Situated LE-CPSs

In this section we formalise cyber-physical systems with learning enabled components (LECs) that are implemented via NNs. At the core of our model is the concept of a group of autonomous LECs situated in an environment, performing actions onto this environment and observing and reacting to their effects [11]. Differently from traditional CPSs, where a control component is often specified in a conventional programming language, we here consider decision-making mechanisms synthesised (or, learned) from data and implemented via NNs. The model that we present in this report generalises the one put forward in [2] for a single autonomous LEC situated in a fully observable environment, and aims to account for compositions of multiple LECs in a partially observable environment. It is defined using an explicit parameter τ for the class of underlying neural networks: ReLU-FFNN, ReLU-RNN, sigmoid-based networks, etc.

We now give a formal description of LE-CPS. A LE-CPS is a closed-loop system composed of an environment and autonomous LECs. We assume that every component's action protocol that governs which actions it can perform at each state of the system has already been synthesised, e.g., via reinforcement learning [9], and is implemented via a neural network. Note that here for simplicity we assume that a LEC has an integrated controller functionality, and thus corresponds to the two components within the grey dashed area in Figure 1. The environment represents

the actual physical environment the LECs are situated in, and corresponds to the above three components in Figure 1.

We consider a set of k LECs $[k] = \{1, \dots, k\}$ acting in an environment E . We treat the environment E as a special component 0, and therefore we consider a system composed of the set $CMP = \{0, 1, \dots, k\}$ of components. Each component (including the environment) can only partially observe the current state of the system, and depending on the type of the underlying NN, has a local state that either simply reflects the observation of the current state of the system, or stores a sequence of observations until the current time point. Since, in general, a component cannot observe the full state of the system, the effect of any of its actions cannot be deterministically determined.

Definition 3.1 (Component). *Consider $CMP = \{0, \dots, k\}$. A synchronous component i , $i \in CMP$, is defined as a tuple $(L_i, Act_i, \text{prot}_i, \mathbf{t}_i)$, where:*

- $L_i \subseteq (\mathbb{R}^{m_i})^*$ is a set of local states for component i , where each local state $l_i \in L_i$ is a sequence of observations from $\mathbb{R}^{m_i}$;
- $Act_i \subseteq \mathbb{R}^{n_i}$ is a set of possible actions for component i ;
- $\text{prot}_i : L_i \rightarrow 2^{Act_i}$ is a local protocol function for component i . The protocol function determines the set of possible actions the component may perform given the current local state.
- $\mathbf{t}_i : L_i \times Act_0 \times \dots \times Act_k \rightarrow L_i$ is a deterministic local transition function that determines the temporal evolution of the component's local states.

The main purpose of this framework is to formalise LECs implemented via NNs. To this end, for $i \in [k]$, we assume that prot_i is implemented via a neural network N_i and so $\text{prot}_i = f_{N_i}$.

We denote by S the set $L_0 \times \dots \times L_k$ of possible global states for the system and by ACT the set $Act_0 \times \dots \times Act_k$ of possible joint actions. For a global state $s = (l_0, \dots, l_k)$ and a component i , we write $l_i(s)$ to denote the local state l_i of component i in s . For a local state l , we write $\text{tail}(l)$ to denote the last observation of l . Intuitively, $\text{tail}(l)$ is all information about the current state of the system available to a component. For an observation $o_i = \text{tail}(l_i) = (c_1, \dots, c_{m_i})$ we use $o_{i,j}$ to denote its j -th component c_j .

Given the above we can now define a learning-enabled cyber-physical system comprising of LECs interacting with an environment.

Definition 3.2. *Let τ be the class of neural networks considered. A learning-enabled cyber-physical system τ -LE-CPS is a tuple*

$$S = ((L_i, Act_i, \text{prot}_i, \mathbf{t}_i)_{i \in [k]}, (L_0, Act_0, \text{prot}_0, \mathbf{t}_0), I)$$

where $I \subseteq L_0 \times \dots \times L_k$ is a set of initial global states, and each prot_i is implemented via NNs of type τ .

Note that this framework assumes stateful components to accommodate LECs based on recurrent neural networks (RNNs). LECs based on RNNs maintain a local state that keeps a history of previous local states, see Section 4.3 for a more detailed discussion. While in the case where LECs are implemented via feed-forward NNs, the local components' states are simply observations of the environment state, and so the framework can be simplified; see Sections 4.2 and 4.5 for more details.

With each LE-CPS $\mathcal{S}$ we can associate a temporal model $\mathcal{M}_{\mathcal{S}}$ that represents all possible evolutions of $\mathcal{S}$.

Definition 3.3 (Synchronous model). *Given a LE-CPS*

$$\mathcal{S} = ((L_i, Act_i, \text{prot}_i, \mathbf{t}_i)_{i \in [k]}, (L_0, Act_0, \text{prot}_0, \mathbf{t}_0), I)$$

of synchronous components, its associated temporal model is a pair $\mathcal{M}_{\mathcal{S}} = (S, T, I)$, where:

- $S = L_0 \times \dots \times L_k$ is the set of global states.
- $T \subseteq S \times ACT \times S$ is the global transition relation defined as $(s, a, s') \in T$ iff $t_i(l_i(s), a) = l_i(s')$, for all $i \in \text{CMP}$.

An $\mathcal{M}_{\mathcal{S}}$ -path, or simply path, is an infinite sequence of global states $s_1 s_2 \dots$ such that for all $j \geq 1$, we have that $(s_j, a_j, s_{j+1}) \in T$ for some global action $a_j \in ACT$.

4 Framework Instantiations

In the rest of the document we illustrate various instantiations of the general framework presented here and accompany them with examples. These impose restrictions on the protocol functions of the LECs, the neural networks implementing the components, and the number of LECs composing the systems. As a result the following describe simpler, less expressive classes of systems that may potentially enable the derivation of more effective verification techniques.

4.1 Asynchronous LE-CPSs

The semantics of LE-CPS assumes that actions are performed at the same round by all the LECs, thereby allowing for a natural modelling of synchronous systems. In this Section we report a fragment of LE-CPS where components evolve asynchronously in an interleaved fashion by insisting on only one local action to be performed at each global transition. Many components can participate in the transition, but when they do, they perform the same local action, thereby synchronising at that particular round; components not participating in the global transition are assumed to perform a special “null” action. So, the interaction patterns of the components can be modelled by means of shared actions, i.e., actions admitted in the repertoire of actions of at least two components. Several settings where autonomous systems are deployed and components operate in accordance to internal instead of global clocks can be modelled in this way.

We now give a formal description of asynchronous LE-CPS. We begin with the definition of an *asynchronous component* that restricts Definition 3.1 to include null actions, and then define then define an asynchronous LE-CPS as a composition of these with an environment and given their temporal evolution.

Definition 4.1 (Asynchronous Component). *An asynchronous component is defined as in Definition 3.1, but additionally adheres to the following:*

- $\epsilon_i \in Act_i$; its repertoire of actions includes a null action ϵ_i .
- for each $l_i \in L_i$, $\epsilon_i \in \text{prot}_i(l_i)$; the null action is enabled in every local state.
- for each $l_i \in L_i$, $t_i(l_i, (\dots, \epsilon_i, \dots)) = l_i$; its state remains the same whenever it performs the null action.

An asynchronous LE-CPS is a closed-loop system comprising asynchronous LECs interacting with an environment.

Definition 4.2. Let τ be the class of neural networks considered. An asynchronous learning-enabled cyber-physical system τ -LE-CPS is a tuple

$$\mathcal{S} = ((L_i, Act_i, \text{prot}_i, \mathbf{t}_i)_{i \in [k]}, (L_0, Act_0, \text{prot}_0, \mathbf{t}_0), I)$$

where each $(L_i, Act_i, \text{prot}_i, \mathbf{t}_i)$ is an asynchronous component, $I \subseteq L_0 \times \dots \times L_k$ is a set of initial global states, and each prot_i are implemented via NNs of type τ .

For each LE-CPS $\mathcal{S}$ we can associate a temporal model $\mathcal{M}_{\mathcal{S}}$ that represents all possible evolutions of the system.

Definition 4.3 (Asynchronous model). Given a LE-CPS

$$\mathcal{S} = ((L_i, Act_i, \text{prot}_i, \mathbf{t}_i)_{i \in [k]}, (L_0, Act_0, \text{prot}_0, \mathbf{t}_0), I)$$

of asynchronous components, its associated temporal model is a pair $\mathcal{M}_{\mathcal{S}} = (S, T, I)$ where for

- $S = L_0 \times \dots \times L_k$ is the set of global states.
- $T \subseteq S \times ACT \times S$ is the global transition relation defined as $(s, (a_0, \dots, a_k), s') \in T$ iff there is an action $a \in \bigcup_i Act_i$ such that the following hold:
 - for all i with $a \in Act_i$, $a_i = a$ and $t_i(l_i(s), (a_0, \dots, a_k)) = l_i(s')$.
 - for all i with $a \notin Act_i$, $a_i = \epsilon_i$ and $t_i(l_i(s), (a_0, \dots, a_k)) = l_i(s')$

4.2 Single FFNN-LEC in deterministic environment

Autonomous systems often comprise precisely one deterministic controller acting in a deterministic environment [2]. For instance the *inverted pendulum* [3] is composed of a pendulum and a controller that aims to keep the pendulum upright by applying a certain force. In these settings the environment (the pendulum and the physical laws) is stateful and updates its state (the position and the angular velocity of the pendulum) in response to the action of the controller, while the controller is stateless but can fully observe the state of the environment, and can perform an action according to its protocol and the state of the environment.

Formally an FFNN-LE-CPS composed of precisely one deterministic LEC and deterministic environment is defined as a tuple

$$((Act, \text{prot}), (S, t_E), I),$$

where

- $S \subseteq \mathbb{R}^m$ for some natural m is the set of states of the environment;
- $Act \subseteq \mathbb{R}^n$ for some natural n is the set of actions of the LEC;
- $\text{prot}: S \rightarrow Act$ is the deterministic protocol function of the LEC that maps S to Act ;
- $t_E: S \times Act \rightarrow S$ is the environment's deterministic transition function that returns the environment's next state given its current one and the action of the LEC.

Distribution (A) Approved for public release; distribution is unlimited.

Note that by the above definition the local state of the environment gives the global system of the system. This is fully observable by the LEC and helps determine the next action the LEC will perform. Also note that, differently from the non-deterministic model described in the previous Section, the temporal model $\mathcal{M}_S$ for deterministic LE-CPS is linear in the sense that for each $s \in I$ there is exactly one path originating from s . The path can be specified by the environment's transition function which can here be viewed as the system's global transition function.

We now give some examples of the above fragment of CPSs.

Example 4.1. Consider the TA4 use case whereby an airplane automatically derives the cross track error while navigating a runway. The system comprises a CNN-based LEC ($\mathbb{R}, \mathbb{R}^{\alpha \times \beta \times 3} \rightarrow \mathbb{R}$) that takes as input an image $im \in \mathbb{R}^{\alpha \times \beta \times 3}$ (where $\alpha \times \beta$ is the size of the image over 3 colour bands) of the runway from cameras attached to the plane, and outputs an estimation $\text{prot}(im)$ of the cross track error.

Example 4.2. Consider the OpenAI Gym [4] task PENDULUM-v0. The system is composed of a pendulum and a LEC which can apply a force to the pendulum. The LEC can observe the current angle of the pendulum (where an angle of zero indicates that it is perfectly vertical) along with the pendulum's angular velocity. Based on this, the LEC chooses an action, which consists of a torque (rotational force) to be applied to the pendulum. The aim of the LEC is to get the pendulum to an upright position and maintain it there.

We formalise the environment for this problem as $E_{\text{pen}} = (S_{\text{pen}}, t_{\text{pen}})$ where:

- S_{pen} is the set of tuples $(\theta, \dot{\theta}) \in [-\pi, \pi] \times [-8, 8]$ where:
 - θ represents the pendulum's angle, and
 - $\dot{\theta}$ represents pendulum's angular velocity.
- $t_{\text{pen}} : S_{\text{pen}} \times \text{Act}_{\text{pen}} \rightarrow S_{\text{pen}}$ is the transition function mapping the state-action pair $((\theta, \dot{\theta}), a)$ to a new state $(\theta', \dot{\theta}')$ (see below for a description of the LEC), where:
 - $\theta' = \theta + \dot{\theta} \cdot dt$, and
 - $\dot{\theta}' = \dot{\theta} + (-(3g/2l) \sin(\theta + \pi) + 3(\max(\min(a, 2), -2))/(ml^2)) \cdot dt$,

where $g = 10$, $m = l = 1$ and $dt = 1/20$. This function is obtained by restricting a to the range $[-2, 2]$ (which is the maximum rotational force that may be applied), and then using the equations of motion for the physical dynamics governing the pendulum.

The LEC responsible for keeping the pendulum upright is formalised as $(\text{Act}_{\text{pen}}, \text{prot}_{\text{pen}})$, where $\text{Act}_{\text{pen}} = \mathbb{R}$, and $\text{prot}_{\text{pen}}((\theta, \dot{\theta}))$ returns $a \in \mathbb{R}$, a force to be applied to the pendulum. We assume that prot_{pen} is implemented via a FFNN that has been suitably trained, and whose weights and biases are fixed.

The set I_{pen} of initial states consists of the single state $(0, 0.1)$ describing the pendulum at zero angle (vertical) and beginning to rotate clock-wise with a small initial angular velocity of 0.1.

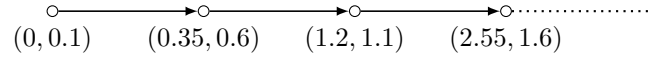
Example 4.3. Assume a 1kg cart on a frictionless track of length 10. The state S_{cart} of the system is given by the set of tuples $(p, u) \in [0, 10] \times \mathbb{R}$, which denotes the position of the cart and its

current velocity. The LEC C_{cart} chooses a force a (in Newtons) in the range $[-1, 1]$ to apply to the cart for 1 second. We define the transition function t_{cart} as follows:

$$t_{cart}((p, u), a) = \begin{cases} (p + u + a/2, u + a) & \text{if } 0 \leq p' \leq 10 \\ (10, 0) & \text{if } p' > 10 \\ (0, 0) & \text{if } p' < 0, \end{cases}$$

where $p' = p + u + a/2$. Observe that t_{cart} is linearly-definable. Assume the initial state set $I_{cart} = \{(0, 0.1)\}$. The resulting system $S_{cart} = (C_{cart}, (S_{cart}, t_{cart}), I_{cart})$ is a linearly-definable LE-CPSs.

Example 4.4. Consider $S_{cart} = (C_{cart}, (S_{cart}, t_{cart}), I_{cart})$ from Example 4.3. Assume the LEC's protocol function $\text{prot}(s) = 0.5$ for all $s \in S_{cart}$, which applies a constant force of 0.5. Then $\mathcal{M}_S$ consists of precisely one path π whose first 4 states are as follows.



4.3 Recurrent LE-CPSs

As a second instantiation we develop a single stateful LEC operating in a deterministic environment [1]. The LEC is implemented by an RNN selecting the next action on the basis of a sequence of possibly incomplete observations of the environment; that sequence constitutes the component's local state. More specifically, if we assume that at each time instant j component i observes $o_j \in \Delta^{m_i}$, then under perfect recall the local state l_i at time instant j is the sequence $o_1 \cdots o_j$.

Formally an RNN-LE-CPS composing precisely one deterministic LEC and deterministic environment is defined as a tuple

$$((L, Act, \text{prot}, t), (L_E, t_E), I),$$

where

- $L_E \subseteq \mathbb{R}^m$ for some natural m is the set of states of the environment;
- $L \subseteq L_E^*$ is the set of states of the LEC;
- $Act \subseteq \mathbb{R}^n$ for some natural n is the set of actions of the LEC;
- $\text{prot} : L \times Act$ is the deterministic protocol function of the LEC that maps L to Act .
- $t_E : L_E \times Act \rightarrow L_E$ is the deterministic environment transition function that computes the next environment state given the current environment state and the action of the LEC.
- $t : L \times Act \rightarrow L$ is the LEC's transition function that computes its next local state given its current one and its action, and is defined by $t(l, a) = l \cdot t_E(\text{tail}(l), a)$.

Following the above definition a local state of the LEC is a sequence of observations of environment states. For simplicity, we assume that the LEC can fully observe the environment. Note that the extension to partial observability (that does not lead to non-determinism) is straightforward and is omitted here for the ease of presentation.

An example of an RNN-LE-CPS would be similar to the example of an FFNN-LE-CPS. The overall setup can be for instance, as in Example 4.2, with the difference that the LEC is implemented via an RNN and, consequently, is able to know the previous positions and velocities of the pendulum.

4.4 LECs operating in non-deterministic environment

As a further class of LE-CPS we consider autonomous systems comprising exactly one deterministic LEC that is deployed in complex and unpredictable environments. To account for the unpredictability of the environment we here define LE-CPSs on partially observable environments. In this setting a LEC cannot observe the full state of the environment and, consequently, the effect of any of its actions on the environment cannot be deterministically defined. For instance, a scout autonomous LEC-drive robot may be visiting a dangerous terrain with unknown obstacles; the reaction of these during the exploration can be modelled via the non-determinism of the environment.

Formally, an FFNN-LE-CPS composing precisely one deterministic LEC and non-deterministic environment is defined as a tuple

$$((Act, \text{prot}), (S, Act_E, \text{prot}_E, \text{t}_E), I),$$

where

- $S \subseteq \mathbb{R}^m$ for some natural m is the set of states of the environment;
- $Act \subseteq \mathbb{R}^n$ for some natural n is the set of actions of the LEC;
- $\text{prot}: S \rightarrow Act$ is the deterministic protocol function of the LEC that maps S to Act ;
- $Act_E \subseteq \mathbb{R}^{n'}$ for some natural n' is the set of actions of the environment;
- $\text{prot}_E: L_E \rightarrow 2^{Act_E}$ is the environment protocol function mapping environment states to subsets of Act_E . Each of these subsets represents the possible reactions of the environment to a LEC's action, and therefore models the non-deterministic behaviour of the environment;
- $\text{t}: S \times Act \times Act_E \rightarrow S$ is the environment's deterministic transition function that computes the next environment state given the current environment state, the action performed by the LEC, and the action performed by the environment.

Example 4.5. Consider an agent navigating in a grid world consisting of walkable areas (ice) and non-walkable areas (ice-holes). The agent can move in any of the four directions (left, down, right and up) with the aim of finding a path along the walkable area to a goal tile. An example of a possible world is depicted below.

S	F	F	F	(S: Starting point, safe)
F	F	F	H	(F: Frozen surface, safe)
F	H	F	H	(H: Hole, agent falls)
H	F	F	G	(G: Goal)

Distribution (A) Approved for public release; distribution is unlimited.

Ice is slippery, which is modelled by non-determinism of the environment: when the agent attempts to walk in a certain direction it could randomly move in that direction or any of the adjacent directions. For instance, if the action chosen by the agent is to move down, then either agent moves down for one step, or the environment forces the agent to move left or right.

We formalise the agent as $Ag_{\mathcal{F}} = (Act, \text{prot})$ where

- $Act = \{0, 1, 2, 3\}$ and 0 stands for left, 1 for down, 2 for right and 3 for up;
- prot is the protocol function implemented via a FFNN network that has been suitably trained.

We formalise the environment for this problem as $E_{\mathcal{F}} = (S_{\mathcal{F}}, Act_E, \text{prot}_E, \mathbf{t}_{\mathcal{F}})$ where:

- $S_{\mathcal{F}}$ is the set $\{1, 2, \dots, 16\}$
- $Act_E = \{-1, 0, 1\}$
- $\text{prot}_E(s) = Act_E$ for each $s \in S_{\mathcal{F}}$
- $\mathbf{t}_{\mathcal{F}}(s, a, 0) = \begin{cases} \text{cell}(r, \min(c-1, 1)), & \text{if } a = 0, \\ \text{cell}(\max(r+1, 4), c), & \text{if } a = 1, \\ \text{cell}(r, \max(c+1, 4)), & \text{if } a = 2, \\ \text{cell}(\min(r-1, 1), c), & \text{if } a = 3, \end{cases}$ and $\mathbf{t}_{\mathcal{F}}(s, a, d) = \mathbf{t}_{\mathcal{F}}(s, a+d \bmod 4, 0)$,

where $r = \lfloor s/4 \rfloor + 1$ and $c = (s-1) \bmod 4 + 1$ are the row and the column of s (in the range from 1 to 4), and given a row r and a column c , the function $\text{cell}(r, c) = 4(r-1) + c$ computes the cell number.

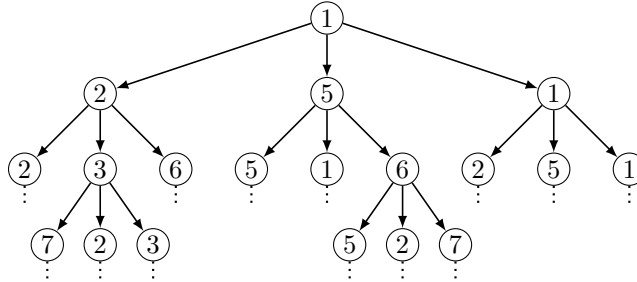
The set of initial states consists of the starting point, $I_{\mathcal{F}} = \{1\}$, and the frozen lake LE-CPS is $S_{\mathcal{F}} = (Ag_{\mathcal{F}}, E_{\mathcal{F}}, I_{\mathcal{F}})$.

An important difference of this class of LE-CPS with the fully deterministic ones of Sections 4.2 and 4.3, is that their temporal evolution is branching instead of linear, as the following example demonstrates.

Example 4.6. *Consider the LE-CPS in Example 4.5. Suppose that*

$$\{(1, 1), (2, 2), (3, 0), (5, 3), (6, 3), (7, 0)\} \subset \text{prot},$$

that is, in state 1 agent wants to go down, in state 2 right, state 3 left, etc. Then, we can depict a portion of $\mathcal{M}_{\mathcal{S}}$ as follows, where the left child always corresponds to the environment action -1, the middle child to action 0, and the right child to action 1:



4.5 Multiple FFNN-LECs

Finally, we present CPSs composed of multiple FFNN-LECs. In order for such a setting to non-trivially extend the single LEC case, we ought to consider LECs with non-deterministic policies, in which case we are able, for instance, to verify whether a group of components can collaborate to achieve a certain goal. It comes at no cost to include a non-deterministic environment, therefore we can define the class FFNN-LE-CPS of cyber-physical systems $\mathcal{S}$ of the form

$$\mathcal{S} = ((Act_i, \text{prot}_i)_{i \in [k]}, (S, Act_0, \text{prot}_0, \mathbf{t}), I)$$

where

- the global system state is the state of component 0, and $S \subseteq \mathbb{R}^m$ for some natural m ;
- LEC i 's local state is simply a (full) observation of the environment state, $i \in [k]$;
- $Act_i \subseteq \mathbb{R}^{n_i}$ for some natural n_i , $i \in \{0, \dots, k\}$;
- prot_i is the component i protocol function mapping S to 2^{Act_i} , $i \in \{0, \dots, k\}$. For each $s \in S$, $\text{prot}_i(s)$ represents the actions that LEC i can take at state s ;
- $\mathbf{t} : S \times Act_0 \times \dots \times Act_k \rightarrow S$ is the global deterministic transition function that computes the next environment state given the current environment state and a tuple of actions of all LECs.

5 Conclusions

In this deliverable we put forward a formal model for LE-CPSs, cyber-physical systems with learning enabled components (LECs). The model accounts for a number of LECs interacting with each other and the environment that they are situated in. The model is intended to be generic so as to be able to represent various kinds of LE-CPSs, including those of practical importance to TA4 performers. Examples of these include: a single LEC operating in a static environment (e.g., an autonomous plane tracking a line), and multiple LECs in a non-deterministic and partially observable environment.

The model is equipped with a formal semantics that allows for reasoning about the temporal evolution of LE-CPSs. As such it lays the appropriate formal background for:

- identifying the formal specification classes covering the TA4 performers' use-cases,
- developing verification methods and algorithms, and
- designing and implementing the Venus verification toolkit.

The technical details of these will be covered in other deliverables.

References

- [1] M. E. Akintunde, A. Kevorchian, A. Lomuscio, and E. Pirovano. Verification of RNN-based neural agent-environment systems. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI19)*. AAAI Press, 2019. To Appear.

- [2] M. E. Akintunde, A. Lomuscio, L. Maganti, and E. Pirovano. Reachability analysis for neural agent-environment systems. In *Proceedings of the 16th International Conference on Principles of Knowledge Representation and Reasoning (KR18)*, pages 184–193. AAAI Press, 2018.
- [3] C. Anderson. Learning to control an inverted pendulum using neural networks. *IEEE Control Systems Magazine*, 9(3):31–37, 1989.
- [4] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. OpenAI Gym. *CoRR*, abs/1606.01540, 2016.
- [5] S. S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 1999.
- [6] Dongsung Huh and Emanuel Todorov. Real-time motor control using recurrent neural networks. In *Proceedings of the IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL09)*, pages 42–49, 2009.
- [7] Scott A. Nivison and Pramod P. Khargonekar. Development of a robust deep recurrent neural network controller for flight applications. In *Proceedings of the American Control Conference (ACC17)*, pages 5336–5342, 2017.
- [8] I. Sutskever, J. Martens, and G. E. Hinton. Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML11)*, pages 1017–1024. Omnipress, 2011.
- [9] R. S. Sutton and A. G. Barto. *Reinforcement Learning – An Introduction*. MIT Press, 1998.
- [10] Yu Wang. A new concept using LSTM neural networks for dynamic system identification. In *Proceedings of the American Control Conference (ACC17)*, pages 5324–5329, 2017.
- [11] M. Wooldridge. *An introduction to MultiAgent systems*. Wiley, second edition edition, 2009.
- [12] A. Zell, N. Mache, R. Hübner, G. Mamier, M. Vogt, M. Schmalzl, and K-U. Herrmann. *SNNS (Stuttgart Neural Network Simulator)*, chapter 9. Springer, 1994.

Identification of target neural network architectures and activation functions with respect to verification needs

Elena Botoeva

Panagiotis Kouvaros

Alessio Lomuscio

1 Introduction

This report identifies neural network architectures for Cyber-Physical Systems (CPSs) that are amenable to formal verification.

Different neural network architectures implement different types of functions $f(\mathbf{x}; \theta)$ that are parameterised over a set of parameters θ . The values of the parameters are learned during a *training phase* so as to give the best possible approximation of some function $f^*(\mathbf{x})$. In the context of CPSs $f^*(\mathbf{x})$ is typically the control logic of the system or a perception component from which the control component takes its input.

Given a neural network implementation $f(\mathbf{x}; \theta)$ of $f^*(\mathbf{x})$ the verification problem concerns establishing whether the output of $f(\mathbf{x}; \theta)$ is always within a safety region w.r.t. a given bounded set of inputs [1, 10, 11, 9, 5]. So, any verification attempt must involve the computation of the network's reachable output set. As a result, the amenability of neural networks to solving their verification problem can be assessed in two ways.

First, the type of operations that $f(\mathbf{x}; \theta)$ comprises determines the type of the verification that can be carried out. If the computation of $f(\mathbf{x}; \theta)$ involves only linear or piecewise linear operations, then *complete verification* is applicable, i.e., the network's output can be determined precisely for a given set of inputs. On the other hand, if the computation of $f(\mathbf{x}; \theta)$ involves non-linear but not piecewise linear operations, then only *incomplete verification* is applicable whereby over-approximations of the network's output are determined.

Second, the architecture of the network does not only influence the type of verification that can be carried out, but also the scalability of the methods employed to solve the verification problem. In particular, most verification procedures are more effective for neural networks with fewer non-linear operations than for those with many.

On the one hand, incomplete methods are typically more scalable than complete ones. This however comes at the cost of often incorrectly giving a negative answer to the verification problem on the basis of over-approximations that span unsafe regions when the actual output of the network is contained within the safety region. On the other hand, given enough time, complete methods always give a correct answer to the verification problem. Still, complete methods are not scalable for big networks. Consequently, there is a trade-off between the two types of methods, and it is therefore a usual verification strategy to involve both kind of methods, where the incomplete ones initially lead the verification process, and the complete ones take over whenever the verification problem cannot be decided.

In the rest of the report we give a brief summary of the most popular neural network architectures, identify whether complete/incomplete verification can be used and comment on the

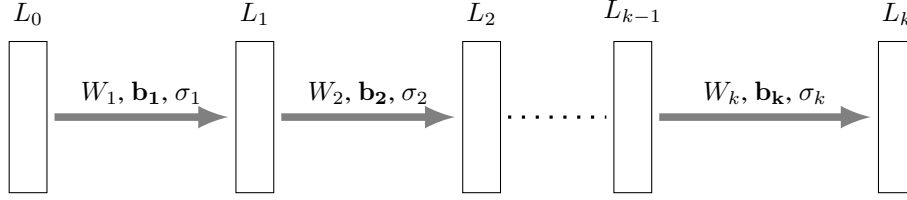


Figure 1: The structure of a feed-forward neural network.

scalability of methods targeting their verification problem.

2 Neural networks architectures

We begin with a discussion on feed-forward neural networks which typically implement stateless controllers in Section 2.1 [4, 7]. In Section 2.2, we present convolutional neural networks which are commonly used for the implementation of image classifiers [6, 2]. Finally, in Section 2.3 we outline recurrent neural networks which are widely used as stateful controllers [12, 3].

2.1 Feed-forward neural networks.

Feed-forward neural networks are directed acyclic graphs whose nodes are structured in layers. The first layer is said to be the *input layer*, the last layer is referred to as the *output layer*, and every layer in-between is called a *hidden layer*. The connections between the nodes are associated with weights which are learnt during the training phase. Figure 1 gives a schematic representation of the structure of a feed-forward neural network.

Given an input vector, the output of the network is computed by propagating the input through the layers of the network. The output of a node at each step is the result of the application of a non-linear activation function to the pre-activation (i.e., the weighted sum of the outputs of the nodes from the previous layer) of the node.

More precisely, for $i \geq 1$ the output $\mathbf{x}_i$ of layer i is the output of the function $f_i: \mathbb{R}^{s_{i-1}} \rightarrow \mathbb{R}^{s_i}$ which is defined by $f_i(\mathbf{x}_{i-1}) \triangleq \sigma(W_i \mathbf{x}_{i-1} + \mathbf{b}_i)$, where σ is a non-linear function which is applied element-wise. A neural network of k layers is then defined as a function $f: \mathbb{R}^{s_0} \rightarrow \mathbb{R}^{s_k}$. The function corresponds to the composition of the functions f_i 's computed by each layer i , i.e., $f(\mathbf{x}) \triangleq f_k(\dots f_1(\mathbf{x}) \dots)$.

Among the most popular activation functions is the *Rectified Linear Unit (ReLU)*. It is defined by $\text{ReLU}(x) \triangleq \max(x, 0)$ for $x \in \mathbb{R}$. This is a piecewise linear function; therefore, ReLU-based feed-forward neural networks are amenable to complete verification. For networks whose output denotes a probability distribution, the *sigmoid* function $\phi(x) \triangleq \frac{1}{1+e^{-x}}$ and *hyperbolic tangent* $\tanh(x) \triangleq \frac{e^{2x}-1}{e^{2x}+1}$ are commonly used. Verification for networks employing these functions typically involves linear approximations of them and it is therefore incomplete. Note that linear approximations can also be used for ReLUs to improve the verification times. This of course comes at the cost of false-negatives which any incomplete verification procedure may produce.

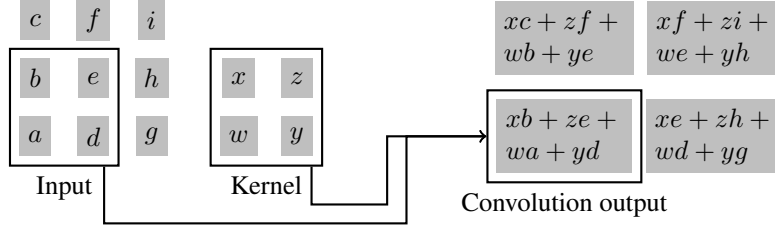


Figure 2: Graphical representation of the convolution operation.

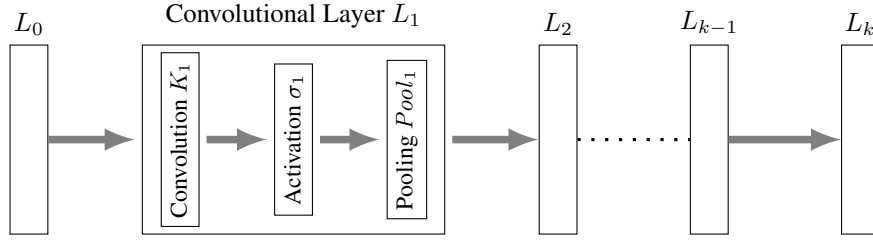


Figure 3: The structure of a convolutional neural network.

2.2 Convolutional neural networks

Learning-enabled components for perception are typically image classifiers implemented as CNNs. CNNs are defined as feed-forward neural networks, but with at least one of the layers comprising a convolution operation instead of the weighted-sum. A convolution is parameterised by two arguments: the input and the kernel. For a two-dimensional input I and a two-dimensional kernel K of size $m \times n$, the operation is defined by

$$\text{Conv}(x, y) = \sum_{j=1}^n \sum_{i=1}^m K(i, j) \cdot I(x + i, y + j).$$

The operation is exemplified in Figure 2. Graphically $\text{Conv}(x, y)$ can be depicted as a node that is connected to all and only the nodes from the previous layer that correspond to the part $(I(x', y'): x \leq x' \leq x + m, y \leq y' \leq y + n)$ of the input. So, differently from feed-forward layers, where a node is connected to all the nodes from the previous layer, convolutional layers exhibit sparse connectivity whenever the size of the kernel is small w.r.t the size of the previous layer. As a result, convolutional layers comprise fewer parameters that need to be determined during the training phase, thereby requiring reduced memory and computational resources. This also affects the complexity of the verification problem which becomes easier to decide.

A convolutional layer typically comprises a convolution operation, followed by a non-linear activation function, followed by a pooling function (see Figure 3).

The pooling function reduces rectangular neighborhoods of the input to summary statistics. The statistics typically used are the *maximum* value, the *average* value or a *norm of the neighbourhood*. Figure 4 exemplifies the application of the max pooling function. In the context of perception systems for CPSs, pooling contributes to the invariance of the network to translations of the input, thereby improving its robustness. Regarding the efficacy of verification procedures,

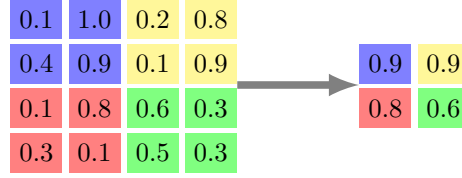


Figure 4: Graphical representation of the max-pooling operation.

architectures that make use of the average pooling function are preferable to those that use max-pooling of norm-pooling functions, since the latter are piecewise linear or non-linear (instead of linear), thereby making the verification problem harder to decide.

2.3 Recurrent neural networks.

Recurrent neural networks (RNNs) are a class of neural networks with cyclic connections between the nodes. Differently from FFNNs, whose output is always the same given a certain single input, RNNs are designed to process sequences of data, and are equipped with a state that evolves over time. As a result, the behaviour of an RNN depends on the history of their inputs. We formally define RNNs below.

A single-layer *recurrent neural network* (RNN) R with h hidden units and input size i and output size o is associated with the weight matrices $W_{i \rightarrow h} \in \mathbb{R}^{i \times h}$, $W_{h \rightarrow h} \in \mathbb{R}^{h \times h}$ and $W_{h \rightarrow o} \in \mathbb{R}^{h \times o}$, and the two activation functions $\sigma : \mathbb{R}^h \rightarrow \mathbb{R}^h$ and $\sigma' : \mathbb{R}^o \rightarrow \mathbb{R}^o$. The weight matrix $W_{i \rightarrow h}$ defines the weights of the connections between input units and hidden units, $W_{h \rightarrow h}$ defines the weights of the connections between hidden units, and $W_{h \rightarrow o}$ defines the weights of the connections between hidden units and output units. We refer to the value of the hidden units as the *state* of the network, due to the self-loop introduced by $W_{h \rightarrow h}$. The activation function σ is applied to the state before it is passed onto the next time step, and σ' is applied to the state of the network at a given time step, in order to compute the output of the network at that time step. We can schematically depict the structure and the connections of an RNN as shown in Figure 5.

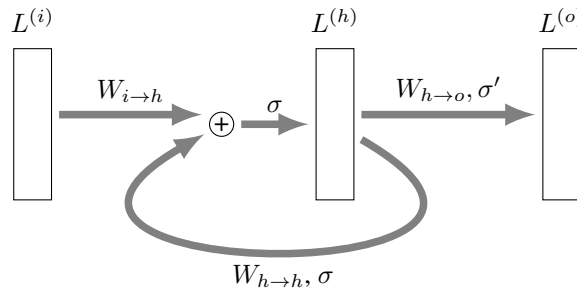


Figure 5: The structure of a recurrent neural network.

Given an input sequence $\mathbf{x} \in (\mathbb{R}^i)^n$ of length n , let

$$h_t^{\mathbf{x}} = \sigma(W_{h \rightarrow h} h_{t-1}^{\mathbf{x}} + W_{i \rightarrow h} \mathbf{x}_t),$$

Distribution (A) Approved for public release; distribution is unlimited.

where $t \in \{1, \dots, n\}$ is a time step, x_t denotes the t -th vector of x and $h_0^x = \mathbf{0}$ is a vector of 0s. The function computed by the network given x can then be defined by

$$f_N(x) = \sigma'(W_{h \rightarrow o} h_n^x).$$

The above computation can be visualised for a sequence of length 3 as shown in Figure 6.

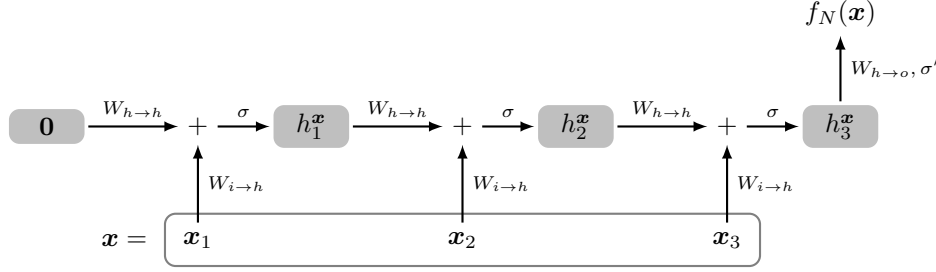


Figure 6: Graphical representation of the function computed by a recurrent neural network.

As with feed-forward neural-networks the activation function can either be piecewise linear, e.g., ReLU, or non-linear, e.g., sigmoid, hyperbolic tangent.

The ability of RNNs to recall information from previous time steps has traditionally made them useful as implementations of stateful controllers for dynamic systems [3, 12, 8]. Currently, the state-of-the-art verification methods for RNNs can only check their behaviour up to a bounded number of time steps. This can be done by unrolling the network into a feed-forward neural network. So, in this setting, recurrent neural networks exhibit the same suitability to verification as with feed-forward neural networks.

3 Conclusions

The deliverable identified feed-forward neural networks, convolutional neural networks, and recurrent neural networks as amenable to verification. Each architecture can be verified exactly by means of complete methods or approximately via incomplete methods. In the case that an architecture includes non-linear components, such as sigmoid activations, then only incomplete verification is applicable.

The attractive properties of CNNs in terms of training carry over to the verification problem. So, perception systems implemented via CNNs can be verified more efficiently than perception systems implemented as feed-forward networks of the same size. Regarding recurrent networks, since they can be unrolled into feed-forward networks, they admit the same verification properties with feed-forward networks. This of course is true only when the verification problem is concerned with checking the behaviour of RNNs up to a bounded time step.

We conclude with the view that the development of both complete and incomplete methods is more beneficial than the development of methods of only one kind. Not only this will enable the verification of architectures including both linear and non-linear components, but it can also enhance the efficacy of the verification of networks with only linear or piecewise linear components. This can be done by combining the better scalability of incomplete methods with the full correctness guarantees that complete methods can provide.

References

- [1] R. Bunel, I. Turkaslan, P. H. S. Torr, P. Kohli, and M. P. Kumar. Piecewise linear neural network verification: A comparative study. *CoRR*, abs/1711.00455v1, 2017.
- [2] A. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*, volume 1. MIT press Cambridge, 2016.
- [3] Dongsung Huh and Emanuel Todorov. Real-time motor control using recurrent neural networks. In *Proceedings of the IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL09)*, pages 42–49, 2009.
- [4] K.J. Hunt, D. Sbarbaro, R. Zbikowski, and P.J. Gawthrop. Neural networks for control systems—A survey. *Automatica*, 28(6):1083–1112, 1992.
- [5] G. Katz, C. W. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *Proceedings of the 29th International Conference on Computer Aided Verification (CAV17)*, volume 10426 of *Lecture Notes in Computer Science*, pages 97–117. Springer, 2017.
- [6] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems (NIPS12)*, pages 1097–1105. Curran Associates, Inc., 2012.
- [7] Derrick H Nguyen and Bernard Widrow. Neural networks for self-learning control systems. *IEEE Control systems magazine*, 10(3):18–23, 1990.
- [8] Scott A. Nivison and Pramod P. Khargonekar. Development of a robust deep recurrent neural network controller for flight applications. In *Proceedings of the American Control Conference (ACC17)*, pages 5336–5342, 2017.
- [9] G. Singh, T. Gehr, M. Püschel, and P. Vechev. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 3(POPL):41, 2019.
- [10] V. Tjeng, K. Xiao, and R. Tedrake. Evaluating robustness of neural networks with mixed integer programming. *CoRR*, abs/1711.07356, 2017.
- [11] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal security analysis of neural networks using symbolic intervals. In *Proceedings of the 27th USENIX Security Symposium (USENIX18)*, 2018.
- [12] Yu Wang. A new concept using LSTM neural networks for dynamic system identification. In *Proceedings of the American Control Conference (ACC17)*, pages 5324–5329, 2017.

Definition of Specification Classes

Imperial College London

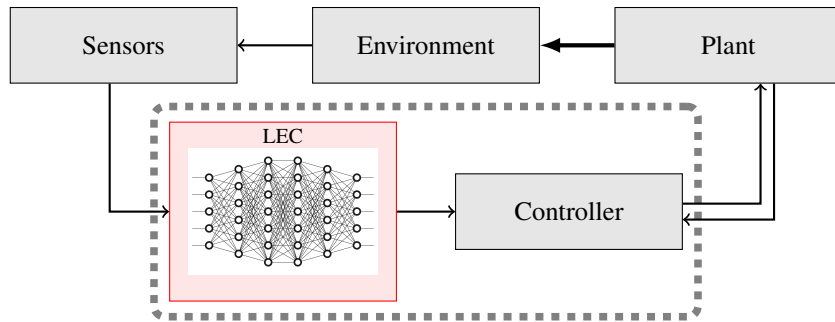
1 Introduction

In the VENUS model of Learning-Enabled Cyber-Physical Systems (LE-CPSs) which formalises Learning-Enabled Components (LECs) operating in an environment and interacting with each other. In this deliverable we define specification classes for reasoning about the behaviour of LE-CPSs. The specifications that we identify can be used to express system-level properties of LE-CPSs, but also local properties concerning the LECs in isolation.

For instance, recall the usual LE-CPSs architecture shown in Figure 1. It can be instantiated to steering systems of autonomous cars and airplanes as follows. The LEC takes as input sensory data and decides the state of the environment, e.g., the cross track error of the runway where the vehicle is navigating. The output of the LEC is passed to a (possibly learning-enabled) controller that operates the steering angle of the vehicle, which in turn changes the state of the environment and the loop is repeated. Then, a system-level property refers to the LE-CPS as a whole, e.g., “the autonomous vehicle stays on the correct track until it has reached its destination”, whereas a local property only concerns the input-output relation of the LEC, e.g., “the cross track error remains unaltered under different light conditions of a certain input”.

We identify three specification classes both for local and for system-level properties. As for local properties of single LECs, the first class can be used to express the reachability (non-

Figure 1: The architecture of LE-CPSs



reachability) of a certain desirable (undesirable) output. The second one captures local properties pertaining to the robustness of the LECs w.r.t. the input variability of their environment. The third one specialises the above class to certain input image transformations and is tailored towards verification of perception neural networks. The system-level properties can be reachability specifications, similarly to the local properties expressing that certain outputs can be obtained, and also more general specifications in temporal logics such as linear-time temporal logic LTL and branching-time temporal logic CTL and their extension alternating-time temporal logic ATL [1]. The last two classes of specifications allow to reason about the temporal evolution of an LE-CPS, and in addition, ATL is equipped with modalities for expressing *strategic abilities* (in the sense that a certain state of affairs can be brought about irrespective of what the opposite coalition is trying to achieve) of LE-CPS-components or coalitions thereof.

The rest of the deliverable is structured as follows. In Section 2 we recall the definition of LE-CPSs and fix the notation used throughout the document. Then, in Section 3 we define the specification classes for verifying local properties of single LECs, and in Section 4 the specification classes for system-level properties of LE-CPSs.

2 Preliminaries

In this section we recall LE-CPS, the model of cyber-physical systems with learning enabled components defined in WP1.1 Month 6 Deliverable. An LE-CPS is a closed-loop system composed of an environment and autonomous LECs. We assume that every component's action protocol that governs which actions it can perform at each state of the system is implemented via a neural network. The environment represents the actual physical environment the LECs are situated in.

We consider a set of k LECs $[k] = \{1, \dots, k\}$ acting in an environment E . We treat the environment E as a special component 0, and therefore we consider a system composed of the set $CMP = \{0, 1, \dots, k\}$ of components. Each component (including the environment) can only partially observe the current state of the system, and depending on the type of the underlying NN, has a local state that either simply reflects the observation of the current state of the system, or stores a sequence of observations until the current time point.

Definition 2.1 (Component). *Consider $CMP = \{0, \dots, k\}$. A synchronous component i , $i \in CMP$, is defined as a tuple $(L_i, Act_i, \text{prot}_i, \text{t}_i)$, where:*

- $L_i \subseteq (\mathbb{R}^{m_i})^*$ is a set of local states for component i , where each local state $l_i \in L_i$ is a sequence of observations from $\mathbb{R}^{m_i}$;
- $Act_i \subseteq \mathbb{R}^{n_i}$ is a set of possible actions for component i ;
- $\text{prot}_i : L_i \rightarrow 2^{Act_i}$ is a local protocol function for component i . The protocol function determines the set of possible actions the component may perform given the current local state.
- $\text{t}_i : L_i \times Act_0 \times \dots \times Act_k \rightarrow L_i$ is a deterministic local transition function that determines the temporal evolution of the component's local states.

We denote by S the set $L_0 \times \dots \times L_k$ of possible global states for the system and by ACT the set $Act_0 \times \dots \times Act_k$ of possible joint actions. For a global state $s = (l_0, \dots, l_k)$ and a component i , we write $l_i(s)$ to denote the local state l_i of component i in s . For a local state l , we write $\text{tail}(l)$ to denote the last observation of l . Intuitively, $\text{tail}(l)$ is all information about

the current state of the system available to a component. For an observation $o_i = \text{tail}(l_i) = (c_1, \dots, c_{m_i})$ we use $o_{i,j}$ to denote its j -th component c_j .

A learning-enabled cyber-physical system comprising of LECs interacting with an environment is then defined as follows.

Definition 2.2. Let τ be the class of neural networks considered (ReLU-FFNN, ReLU-RNN, etc). A learning-enabled cyber-physical system τ -LE-CPS is a tuple

$$\mathcal{S} = ((L_i, \text{Act}_i, \text{prot}_i, \mathbf{t}_i)_{i \in [k]}, (L_0, \text{Act}_0, \text{prot}_0, \mathbf{t}_0), I)$$

where $I \subseteq L_0 \times \dots \times L_k$ is a set of initial global states, and each prot_i is implemented via NNs of type τ .

The temporal model $\mathcal{M}_{\mathcal{S}}$ of a LE-CPS $\mathcal{S}$ that represents all possible evolutions of $\mathcal{S}$ is defined as below.

Definition 2.3 (Synchronous model). Given a LE-CPS

$$\mathcal{S} = ((L_i, \text{Act}_i, \text{prot}_i, \mathbf{t}_i)_{i \in [k]}, (L_0, \text{Act}_0, \text{prot}_0, \mathbf{t}_0), I)$$

of synchronous components, its associated temporal model is a pair $\mathcal{M}_{\mathcal{S}} = (S, T, I)$, where:

- $S = L_0 \times \dots \times L_k$ is the set of global states.
- $T \subseteq S \times ACT \times S$ is the global transition relation defined as $(s, a, s') \in T$ iff $t_i(l_i(s), a) = l_i(s')$, for all $i \in \text{CMP}$.

An $\mathcal{M}_{\mathcal{S}}$ -path, or simply path, is an infinite sequence of global states $s_1 s_2 \dots$ such that for all $j \geq 1$, we have that $(s_j, a_j, s_{j+1}) \in T$ for some global action $a_j \in ACT$. Given a path ρ , we use $\rho(j)$ to denote the j -th state in ρ , $j \geq 1$. For a state s , we use $\text{paths}(s)$ to denote the set of $(\mathcal{M}_{\mathcal{S}})$ -paths starting from s .

3 Specification Classes for Isolated LECs

We identified three specification classes of local properties concerning LECs in isolation.

- (1) Reachability (and safety) properties concerning the realisability of desirable or undesirable states.
- (2) Specifications pertaining to the local robustness of LECs w.r.t. random noise.
- (3) A variant of local robustness w.r.t. geometric and photometric transformations, also referred to as transformational robustness.

Next, we discuss each class in detail.

3.1 Reachability

Reachability is a typical property studied by verification approaches whose goal is to check whether the neural network can output certain values (usually undesirable or erroneous), given the set of possible inputs. For instance, there have been developed a number neural networks producing steering advisories for the unmanned variant of the Airborne Collision Avoidance System X, and in order to certify that they behave as intended, it is necessary to verify that these networks assign correct scores to the output advisories for various input sets. When formulated as a reachability property, the goal then is to find an admissible input that produces output advisories with incorrect scores, or to prove that no such input exists. In the former case, the found witness can be used to augment the training set to retrain the neural network.

We now define reachability formally.

Definition 3 .1. *Given a neural network N computing function f_N , a set of initial values I and a condition C on the output of N , C is reachable via N from I if there exists $i \in I$ such that $C(f_N(i))$ holds.*

Here, the condition C is expressed as a Boolean combination of linear arithmetic constraints.

Safety is the complement of the reachability property that aims at certifying whether a given network is always safe/robust, that is, it never produces undesirable outputs.

Definition 3 .2. *Given a neural network N computing function f_N , a set of initial values I and a condition C on the output of N , (N, I) is safe with respect to C , if $C(f_N(i))$ holds for all $i \in I$.*

Safety is the complement of reachability in the sense that (N, I) is safe w.r.t. C iff $\neg C$ is not reachable via N from I .

3.2 Local Robustness

The second class of specifications is concerned with checking invariance properties of neural networks against small perturbations of their inputs.

We are here considering *local robustness* properties whose aim is to verify that the network is *robust* with respect to a specified class P of perturbations of the original input x , i.e., the network does not distinguish x from its modifications in $P(x)$. Indistinguishability is defined by means of a (Boolean) property on a pair of the network outputs, and can, for instance, state that the argmax values of the two outputs coincide (the two inputs are classified in the same way).

Definition 3 .3. *Given an input x , a perturbation class P and a condition C on a pair of outputs of N , the network N is locally robust for x with respect to P and C if $C(f_N(x), f_N(x'))$ holds for every perturbation $x' \in P(x)$.*

Note that local robustness can be seen as a special case of a safety property when I coincides with $P(x)$.

It is customary to consider the perturbation class defined as a bounded L_p -norm ball of radius r centred around an original input x , denoted $B_r^p(x)$, for $p \in \{1, 2, \infty\}$. Formally, $B_r^p(x)$ includes all points x' that are different from x by no more than r when measured using L_p -norm, i.e., $\|x' - x\|_p \leq r$. In the case of perception systems, such perturbations correspond to adding noise to an image, imperceptible or insignificant to the human eye, and in the case of natural language processing systems, such perturbations correspond to adding typos to a sentence.

3.3 Transformational Robustness

The last class of specifications defines local robustness properties for perception systems but only with respect to certain classes of transformations.

We consider two classes of transformations that are tailored towards a thorough analysis of perception systems, such as autonomous vehicles using convolutional neural networks to perceive and interpret the environment (road signs, traffic lights, other vehicles, etc). They are defined as affine and photometric transformations, capturing the intuition that autonomous vehicles should correctly identify objects irrespective of, e.g., the angle and position of the camera, the environment light conditions, the position of the object in the image, the distance to the object from the camera.

An *affine transformation* is a mapping between affine spaces that preserves collinearity and ratios of distances [3]. It is represented in vector algebra as follows.

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

where (x, y) is a point, $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$ is a non-singular matrix, and $(t_x, t_y)^T$ is the *translation vector*. An affine transformation whereby A equals the identity matrix is said to be a *translation*. An affine transformation is referred to as *scaling* if $A = \sigma I$, $t_x = 0$ and $t_y = 0$ for a scale factor σ . In the case where $\sigma < 1$ the scaling is called *subsampling*, whereas in the case where $\sigma > 1$ the scaling is known as *zooming*.

A *photometric transformation* is an affine change in the intensity of a pixel p [5]. It is defined as $f(p) = \alpha p + \beta$. If $0 < \alpha < 1$, then the transformation reduces the contrast of the image. Otherwise, if $\alpha > 1$, the transformation increases the contrast of the image. The factor β controls the luminosity of the image with higher values pertaining to brighter images.

Given the two classes of transformations above, we define transformational robustness as follows.

Definition 3.4. *Given an input x , a transformation class T and a condition C on a pair of outputs of N , the network N is locally transformationally robust for x with respect to T and C if $C(f_N(x), f_N(x'))$ holds for every transformation $x' \in T(x)$.*

4 Specification Classes for LE-CPSs

We identified three specification classes of system-level properties for LE-CPSs.

- (1) Reachability and safety properties concerning the realisability (non-reachability) of desirable (undesirable) states.
- (2) Properties expressed in linear- and branching-time temporal logics (LTL and CTL) that allow to reason about the temporal evolution of a system,
- (3) Properties expressed in alternating-time temporal logic (ATL), a generalisation of the above logics that allows to reason about strategic abilities of LE-CPS-components or coalitions thereof.

Next, we discuss each class in detail.

4.1 Reachability

Reachability is a typical property being studied not only of isolated LECs, but also of closed-loop systems. For instance, in the context of an airborne collision avoidance system, one might want to verify whether starting from a configuration leading to a collision, it can output a clear-of-conflict. In the case of a positive answer, it is possible to identify an initial state from which the system evolves to produce the undesirable output, and then to use it to “debug” the system (and if needed, to retrain the learning enabled components).

Next, we define two variants of reachability: an unbounded and a bounded one.

Definition 4.1. *Consider an LE-CPS $\mathcal{S}$ with initial state I and a condition C on the global state of $\mathcal{S}$. Then C is reachable via $\mathcal{S}$ if there exists an initial state $i \in I$, a path ρ in $\mathcal{M}_{\mathcal{S}}$ starting from i and a natural number k such that $\rho(k)$ satisfies C .*

Definition 4.2. *Consider an LE-CPS $\mathcal{S}$ with initial state I , a condition C on the global state of $\mathcal{S}$ and a bound d . Then C is reachable via $\mathcal{S}$ in k steps if there exists an initial state $i \in I$, a path ρ in $\mathcal{M}_{\mathcal{S}}$ starting from i and a natural number $k \leq d$ such that $\rho(k)$ satisfies C .*

The first definition corresponds to the unbounded version of reachability as the number of steps which are required to reach C (here, k) is not known in advance. In contrast, in the second definition, a bound d on the maximum number of steps to be considered is given, hence bounded reachability.

Safety is the complement of the reachability property that aims at certifying whether a given system is always safe/robust, that is, it never reaches deadlocks, failures, etc.

Definition 4.3. *Consider an LE-CPS $\mathcal{S}$ with initial state I and a (safety) condition C on the global state of $\mathcal{S}$. Then $\mathcal{S}$ is safe with respect to C if for all paths ρ in $\mathcal{M}_{\mathcal{S}}$ starting from a state $i \in I$ and for all $k \geq 0$, we have that $\rho(k)$ satisfies C .*

Safety is the complement of reachability in the sense that $\mathcal{S}$ is safe w.r.t. C iff $\neg C$ is not reachable via $\mathcal{S}$.

4.2 The Temporal Logics LTL and CTL

We define an expressive class of specifications well adapted for analysing the evolution of closed-loop systems, and capable of expressing reachability and safety properties. To this end, we introduce the languages of linear-time temporal logic LTL [4] and of branching-time temporal logic CTL [2] where atomic propositions are linear constraints on the components of local states. These languages are tailored to being amenable to sound and complete verification of LE-CPSs, for this reason atomic propositions are restricted to linear constraints.

We start with the definition of atomic propositions.

Definition 4.4. *Given local states $L_i \subseteq (\mathbb{R}^{m_i})^*$, $i \in \{0, \dots, k\}$, the (state component) variables x and atomic propositions α are expressions of the following form:*

$$\begin{aligned} x &::= (i.j) \\ \alpha &::= cx + \dots + cx \text{ op } v \end{aligned}$$

where $i \in \{0, \dots, k\}$, $j \in [m_i]$, $c, v \in \mathbb{R}$, and $\text{op} \in \{<, \leq, =, \geq, >\}$.

A variable $x = (i.j)$ refers to the j -th component of the last observation in local state l_i of agent i , i.e., to $\text{tail}(l_i).j$. Thus, e.g., an atomic proposition $(i.j) = v$ states that the value of j -th component of $\text{tail}(l_i)$ is v , while $(i.j) - (i.j') = 0$ states that the j -th and j' -th components of $\text{tail}(l_i)$ are the same. We denote by $\mathcal{A}$ the language consisting of the atomic propositions α . Note that $\mathcal{A}$ allows us to compare the observations of the same time point only.

We now define the grammar and semantics of LTL for a given set of propositions.

Definition 4.5. LTL formulas are defined by the following BNF:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid X\varphi \mid \varphi U \varphi,$$

where $p \in \mathcal{A}$.

As is customary for temporal logics, the operators X (“next”) and U (“until”) are temporal operators. Additional Boolean connectives and temporal operators are defined from $\neg$, $\vee$, X and U in the usual way. In particular, the LTL formula $F\varphi$ (“eventually” φ) is equivalent to $\top U \varphi$, and in turn $G\varphi$ (“globally” φ) can be expressed as $\neg F \neg \varphi$.

LTL formulas are interpreted on the model $\mathcal{M}_S = (G, T, I)$ in a standard way. We write $\rho \models \varphi$ to mean that a formula is true at a path ρ , and $\mathcal{M}_S, s \models \varphi$ to mean that a formula is true at a state s in $\mathcal{M}_S$.

Definition 4.6 (LTL Satisfaction). For a path ρ and an LTL formula φ the satisfaction relation $\models$ is defined inductively as follows:

$$\begin{aligned} \rho \models (\sum_{h=1}^n c_h(i_h.j_h)) \text{ op } v & \text{ if } (\sum_{h=1}^n c_h \cdot o_{i_h}.j_h) \text{ op } v \text{ for } o_{i_h} = \text{tail}(l_{i_h}(\rho[1])); \\ \rho \models \neg\varphi & \text{ if } \rho \not\models \varphi; \\ \rho \models \varphi_1 \vee \varphi_2 & \text{ if } \rho \models \varphi_1 \text{ or } \rho \models \varphi_2; \\ \rho \models X\varphi & \text{ if } \rho[2] \models \varphi. \\ \rho \models \varphi U \psi & \text{ if there exists a position } j \geq 1 \text{ such that } \rho[j, \infty] \models \\ & \psi \text{ and for all positions } 1 \leq j' < j, \text{ we have} \\ & \rho[j', \infty] \models \varphi. \end{aligned}$$

Moreover, for a model $\mathcal{M}_S$ and a global state s , we have $\mathcal{M}_S, s \models \varphi$ if $\rho \models \varphi$ for every path $\rho \in \text{paths}(s)$.

We next proceed to define the grammar and semantics of CTL.

Definition 4.7. CTL formulas are defined by the following BNF:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid AX\varphi \mid AF\varphi \mid E(\varphi U \varphi),$$

where $p \in \mathcal{A}$.

Here A is a universal path quantifier and E an existential path quantifier, and the formula $AX\varphi$ stands for “in all paths φ holds after 1 time step”, $AF\varphi$ for “in all paths φ eventually holds” and $E(\varphi U \psi)$ for “there is a path such that ψ holds within a finite number of time steps, and where φ holds up until then.” Additional CTL temporal modalities are defined from the operators included into the definition in the usual way: $EX\varphi \equiv \neg AX \neg \varphi$, $EF\varphi \equiv E(\top U \varphi)$, $AG\varphi \equiv \neg EF \neg \varphi$, $EG\varphi \equiv \neg AF \neg \varphi$ and $A(\psi U \varphi) \equiv \neg(E((\neg\varphi)U \neg(\psi \vee \varphi)) \vee EG \neg \varphi)$.

CTL formulas are interpreted on the model $\mathcal{M}_S = (G, T, I)$ in a standard way. As with LTL formulas, we write $(\mathcal{M}_S, s) \models \varphi$ to mean that a formula is true at a state s in $\mathcal{M}_S$. If $\mathcal{M}_S$ is clear from the context, then we simplify the notation to $s \models \varphi$.

Definition 4.8 (Satisfaction). *For a model $\mathcal{M}_S$, a global state s , a path ρ , and a CTL formula φ , the satisfaction relation $\models$ is defined inductively as follows:*

$$\begin{array}{ll}
s \models (\sum_{h=1}^n c_h(i_h.j_h)) \text{ op } v & \text{iff } (\sum_{h=1}^n c_h \cdot \text{tail}(l_{i_h}(s)).j_h) \text{ op } v; \\
s \models \neg\varphi & \text{iff } s \not\models \varphi; \\
s \models \varphi_1 \vee \varphi_2 & \text{iff } s \models \varphi_1 \text{ or } s \models \varphi_2; \\
s \models AX\varphi & \text{iff for each path } \rho \in \text{paths}(s), \text{ we have } \rho[2] \models \varphi. \\
s \models AF\varphi & \text{iff for each path } \rho \in \text{paths}(s), \text{ there exists a position} \\
& \quad j \geq 1 \text{ such that } \rho[j] \models \varphi. \\
\pi \models E(\varphi U \psi) & \text{iff there exists a path } \rho \in \text{paths}(s) \text{ and a position} \\
& \quad j \geq 1 \text{ such that } \rho[j] \models \psi \text{ and for all positions} \\
& \quad 1 \leq j' < j, \text{ we have } \rho[j'] \models \varphi.
\end{array}$$

We note that a reachability property can be expressed as a CTL formula $EF\alpha$, where $\alpha \in \mathcal{A}$.

4.3 The Temporal Logic ATL*

We conclude with a final specification class for LE-CPSs that is a generalisation of the languages introduced in the section above. Alternating-time temporal logic ATL [1] is well suited for reasoning about strategic abilities of LE-CPSs components or coalitions thereof.

We now define the grammar of the ATL* language that subsumes both LTL and CTL.

Definition 4.9 (ATL*). *ATL* state formulas φ and path formulas ψ over $\mathcal{A}$ are defined by the following BNFs:*

$$\begin{aligned}
\varphi &::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \langle\langle Co \rangle\rangle\psi, \\
&::= \varphi \mid \neg\psi \mid \psi \vee \psi \mid X\psi \mid \psi U \psi \mid G\psi
\end{aligned}$$

where $p \in \mathcal{A}$ and $Co \subseteq [k]$.

As is customary for ATL and ATL*, the operator $\langle\langle \cdot \rangle\rangle$ is a path quantifier. Satisfaction of state formulas is related to a specific state, while of path formulas to a specific computation (path). The path formula $X\psi$ stands for “ ψ holds in the next step of the path”, while the state formula $\langle\langle Co \rangle\rangle\psi$ intuitively means that “the agents in Co can cooperate to make ψ true”. The dual path quantifier $\llbracket Co \rrbracket\psi$ is defined as $\neg\langle\langle Co \rangle\rangle\neg\psi$ with the intuitive meaning that “the agents in Co cannot cooperate to avoid ψ ”. ATL* allows for an arbitrary combination of temporal operators before being prefixed by a path quantifier. In contrast, similarly to CTL, in an ATL formula, path quantifiers and temporal operators strictly alternate.

ATL specifications are well suited for analysing properties of closed-loop systems composed of multiple components as they allow to reason about what groups of components can achieve by collective action. For instance, one can express properties for verifying if an advisory has a strategy (whatever the rest of the components attempt) to achieve its goal.

The branching-time temporal logic CTL/CTL* is a fragment of ATL/ATL* with only existential E or universal A quantification over paths. The CTL path quantifiers E and A can be defined as abbreviations for $\langle\langle CMP \rangle\rangle$ and $\llbracket CMP \rrbracket$, respectively, where we remind that $CMP = \{0, \dots, k\}$ is the set of all components. LTL is the fragment of ATL* consisting of state formulas of the form $A\psi$, where ψ does not contain path quantifiers $\langle\langle Co \rangle\rangle$.

We are interested in the specifications from the logic ATL*($\mathcal{A}$) that consist of the set of state formulas φ over $\mathcal{A}$. We now define the semantics of ATL*($\mathcal{A}$).

ATL* formulae are interpreted on the model $\mathcal{M}_S = (G, T, I)$ in a standard way. Define a strategy for agent i to be a function $f_i : L_i \rightarrow Act_i$ such that $f_i(l_i) \in \text{prot}_i(l_i)$, i.e., $f_i(l_i)$ is one of the actions enabled for agent i by the protocol prot_i when its local state is l_i . Given a group of agents A and a state $s \in S$, let $F_A = \{f_i \mid i \in A\}$ be a set of strategies. We denote by $\text{out}(s, F_A) \subseteq \Pi(s)$ the set of computations that the group A can enforce at state s , in the following sense. Consider the following game between a protagonist and an antagonist. The game proceeds in an infinite sequence of rounds, and after each round, the position of the game is a state of S . The initial position is s . Now consider the game in some position s' . To update the position, first the protagonist chooses for every player $i \in A$ a move $a_i \in \text{prot}_i(s')$. Then, the antagonist chooses for every player $j \in \text{CMP} \setminus A$ a move $a_j \in \text{prot}_j(s')$, and the position of the game is updated to $(t_0(s', a), \dots, t_k(s', a))$ for $a = (a_0, \dots, a_k)$. In this way, the game continues forever and produces a computation.

We write $(\mathcal{M}_S, s) \models \varphi$ ($(\mathcal{M}_S, \pi) \models \psi$, respectively) to mean that a state formula (path formula, respectively) is true at a state s (path π , respectively) in $\mathcal{M}_S$. If $\mathcal{M}_S$ is clear from the context, then we simplify the notation to $s \models \varphi$ ($\pi \models \psi$, respectively).

Definition 4.10 (Satisfaction). *For a model $\mathcal{M}_S$, a global state s , a path π , an ATL* state formula φ and an ATL* path formula ψ , the satisfaction relation $\models$ is defined inductively as follows:*

$$\begin{aligned}
s \models (\sum_{h=1}^n c_h(i_h \cdot j_h)) \text{ op } v & \text{ iff } (\sum_{h=1}^n c_h \cdot \text{tail}(l_{i_h}(s)) \cdot j_h) \text{ op } v; \\
s \models \neg \varphi & \text{ iff } s \not\models \varphi; \\
s \models \varphi_1 \vee \varphi_2 & \text{ iff } s \models \varphi_1 \text{ or } s \models \varphi_2; \\
s \models \langle\langle A \rangle\rangle \psi & \text{ iff there exists a set } F_A \text{ of strategies, one for each} \\
& \text{ player in } A, \text{ such that for all computations } \pi \in \\
& \text{ out}(s, F_A), \text{ we have } \pi \models \psi. \\
\pi \models \varphi & \text{ iff } \pi[1] \models \varphi. \\
\pi \models \neg \psi & \text{ iff } \pi \not\models \psi; \\
\pi \models \psi_1 \vee \psi_2 & \text{ iff } \pi \models \psi_1 \text{ or } \pi \models \psi_2; \\
\pi \models X\psi & \text{ iff } \pi[2, \infty] \models \psi. \\
\pi \models \psi_1 U \psi_2 & \text{ iff there exists a position } j \geq 1 \text{ such that } \pi[j, \infty] \models \\
& \psi_2 \text{ and for all positions } 1 \leq j' < j, \text{ we have} \\
& \pi[j', \infty] \models \psi_1.
\end{aligned}$$

5 Conclusions

In this deliverable we defined the specification classes for analysing and reasoning about the behaviour of learning enabled components (LECs) in isolation, and of LE-CPSs, cyber-physical systems with LECs. We identified three classes for specifying (local) properties of single LECs:

- reachability and safety,
- local robustness,
- transformational robustness,

and three classes for specifying system-level properties of LE-CPSs:

- reachability and safety,
- temporal logics LTL and CTL, and

- temporal logic ATL^* .

Satisfaction of specifications is determined by relying on a formal semantics. Together with the WP1.1 Month 6 Deliverable this document lays the appropriate formal background for:

- developing verification methods and algorithms, and
- designing and implementing the Venus verification toolkit.

The technical details of these will be covered in other deliverables.

References

- [1] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):672–713, 2002.
- [2] E. A. Emerson and J. Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences*, 30:1–24, 1985.
- [3] R. Hartley and A. Zisserman. *Multiple view geometry in computer vision*. Cambridge university press, 2003.
- [4] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th International Symposium Foundations of Computer Science (FOCS77)*, pages 46–57, 1977.
- [5] M. Sonka, V. Hlavac, and R. Boyle. *Image processing, analysis, and machine vision*. Cengage Learning, 2014.

Methods for Reachability Analysis of ReLU LE-CPSs via MILP

1 Introduction

This report describes sound and complete methods for verifying properties of ReLU-based learning enabled cyber-physical systems (LE-CPSs), both with feed-forward and recurrent learning-enabled components (LECs), via mixed-integer linear programming (MILP). In this deliverable, we focus on the analysis of local and system-level reachability properties. Namely, we are interested in the following two reachability problems:

- Given a neural network N computing function f_N , a set I of initial values and a condition C on the output of N , decide whether there exists $i \in I$ such that $C(f_N(i))$ holds.
- Given an LE-CPS $\mathcal{S}$ with the set I of initial states and a bounded CTL specification φ , decide whether there exists an initial state $s_0 \in I$ such that $\mathcal{M}_{\mathcal{S}}, s_0 \models \varphi$.

Exact formulations of the above problems can be given as mixed integer linear programs. Notably, verifying properties from all specification classes defined in WP1.1 Month 12 Deliverable can be reduced to the above reachability analysis.

We now recall some basics of MILP and how a piecewise linear function can be modelled as a set of mixed integer linear constraints. A *mixed-integer linear program* π is a set of linear constraints over integer and real-valued variables. *Binary* variables are integer-valued variables that can take values 0 or 1. We use $\text{vars}(\pi)$ to denote the set of variables in π . A MILP π is *feasible* if there exists an assignment $a : \text{vars}(\pi) \rightarrow \mathbb{R}$ of values from $\mathbb{R}$ to variables from π that *satisfies* π , denoted $a \models \pi$. In particular, $a \models \pi$ iff $a(\delta) \in \{0, 1\}$, for each binary variable δ , and all constraints in π are satisfied.

It is well known that piecewise linear functions can be encoded as a set of mixed integer linear constraints using the “Big-M” method [5]. For instance, the ReLU function $y = \max(x, 0)$ when x takes values from the interval $[l, u]$ gives rise to the following set of constraints using a binary variable δ , in addition to real-valued variables x and y , and constants l and u :

$$\begin{aligned} y &\geq 0, & y &\leq u \cdot \delta, \\ y &\geq x, & y &\leq x - l \cdot (1 - \delta). \end{aligned}$$

Here, when $\delta = 1$, the constraints imply that $y = x$ and $x \geq 0$, and when $\delta = 0$, the constraints imply that $y = 0$ and $x \leq 0$. We can write the same constraints in an equivalent form making use

of *indicator* constraints (implication clauses of the form $c_1 \Rightarrow c_2$):

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

These constraints are read as follows: if $\delta = 1$ then $x \geq 0$ and $y = x$ should hold, and if $\delta = 0$ then $x \leq 0$ and $y = 0$ should hold.

Recall that the function computed by a feed-forward ReLU-based neural network is piecewise linear [4]. In the rest of this report we assume that given any piecewise linear function $\mathbf{y} = f(\mathbf{x})$, where $\mathbf{y}$ and $\mathbf{x}$ are tuples of variables, there is a MILP program $\Pi_f(\mathbf{x}, \mathbf{y})$ with $\mathbf{x} \cup \mathbf{y} \subset \text{vars}(\Pi_f(\mathbf{x}, \mathbf{y}))$ whose solutions represent the function f . More formally:

- Let $\Pi_f(\mathbf{x}, \mathbf{y})$ be an MILP program corresponding to $\mathbf{y} = f(\mathbf{x})$. Given $u \in \mathbb{R}^m$ and $v \in \mathbb{R}^n$, we have that $v = f(u)$ iff there is an assignment a to $\text{vars}(\Pi_f(\mathbf{x}, \mathbf{y}))$ such that $u = a(\mathbf{x})$, $v = a(\mathbf{y})$, and $a \models \Pi_f(\mathbf{x}, \mathbf{y})$.

The rest of the deliverable is structured as follows. In Section 2 we provide the reachability method for single LECs. Then, in Section 3 we develop a method for reachability analysis for feed-forward LE-CPSs, and in Section 4 show how to augment it to handle recurrent LECs.

2 Reachability analysis for LECs

In this section we present an MILP-based method for checking reachability properties of a learning enabled component considered in isolation.

We thus assume to have a neural network N computing function f_N , a set I of initial values and a condition C on the output of N . Moreover, we assume that both I and C can be encoded as sets of mixed-integer linear constraints. For instance, I can be a union of hyper-rectangles of the form $(l_1, u_1) \times \dots \times (l_m, u_m)$, $l_i, u_i \in \mathbb{R}$, and C can be a negation-free Boolean combination of linear constraints on the output of N (disjunction/union can be encoded with the help of binary/integer variables).

Let $\mathbf{x}$ and $\mathbf{y}$ denote the tuples of variables corresponding to the input and the output of the network, respectively. Denote by $\Pi_I(\mathbf{x})$ the set of linear constraints on $\mathbf{x}$ corresponding to I , and by $\Pi_C(\mathbf{y})$ the set of linear constraints on $\mathbf{y}$ corresponding to C . Then,

Theorem 2.1 ([6]). *C is reachable via N from I , that is, there exists $i \in I$ such that $C(f_N(i))$ holds iff the MIL program $\Pi = \Pi_I(\mathbf{x}) \cup \Pi_{f_N}(\mathbf{x}, \mathbf{y}) \cup \Pi_C(\mathbf{y})$ is feasible. Furthermore, when Π is feasible, let $i = a(\mathbf{x})$ for a satisfying assignment a to $\text{vars}(\Pi)$. Then $i \in I$ and $C(f_N(i))$ holds.*

Thus, we have reduced the reachability problem for a LEC to the feasibility problem of a MILP program. There exist a number of open source and commercial solvers that can effectively solve the latter. The efficiency of the feasibility check depends on how tight the MILP formulation is, and in particular, on how tight the lower and upper bound estimates l and u of ReLU neurons are. Scalability and what can be done to improve verification times will be covered in Phase 2.

We conclude by reminding local robustness of a network can be decided via solving an appropriate reachability problem using the above approach.

3 Reachability analysis for feed-forward LE-CPSs with non-deterministic environment

In this section we present an MILP-based method for checking reachability of bounded CTL specifications for feed-forward LE-CPSs composed of exactly one deterministic LEC that is deployed in a non-deterministic environment, and defined as a tuple $((Act, \text{prot}), (S, t_E), I)$, where

- $S \subseteq \mathbb{R}^m$ for some natural m is the set of states of the environment;
- $Act \subseteq \mathbb{R}^n$ for some natural n is the set of actions of the LEC;
- $\text{prot}: S \rightarrow Act$ is the deterministic protocol function of the LEC implemented via a feed-forward neural network;
- $t_E: S \times Act \rightarrow 2^S$ is a transition function which determines the environment's set of next possible states given its current state and an agent's action, where $t_E(s, a)$ is a finite set for all $s \in S$ and $a \in Act$.

Bounded CTL is the fragment of CTL where all temporal modalities are indexed with natural numbers denoting their temporal depth and which can be defined by the following grammar:

$$\begin{aligned}\varphi &::= \alpha \mid \varphi \vee \varphi \mid EX^k \varphi \mid \varphi \wedge \varphi \mid AX^k \varphi, \\ \alpha &::= (c_1(d_1) + \dots + c_n(d_n)) \text{ op } c\end{aligned}$$

where $\text{op} \in \{\leq, =, \geq\}$, $d_i \in \{1, \dots, m\}$, $c_i, c \in \mathbb{R}$, and $k \in \mathbb{N}$. Here, α denotes atomic formulas that are in general linear inequalities over the components of a state. 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”. Moreover, we can express bounded until with the help of the abbreviations $E(\varphi U^1 \psi) \triangleq \psi \vee (\varphi \wedge EX^1 \psi)$, and $E(\varphi U^k \psi) \triangleq \psi \vee (\varphi \wedge EX^1 E(\varphi U^{k-1} \psi))$ for $k > 1$, and analogously with $A(\varphi U^k \psi)$. The formula $E(\varphi U^k \psi)$ stands for “there is a path such that ψ holds within k time steps, and where φ holds up until then.”

We first note that we can assume without loss of generality that the cardinality of $t_E(s, a)$ is the same for each state s and action a .

Lemma 3.1. *Given a LE-CPS $S = ((Act, \text{prot}), (S, t_E), I)$ and a CTL specification φ , there is a LE-CPS $S' = ((Act, \text{prot}'), (S', t'_E), I')$, 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 φ' such that $S \models \varphi$ iff $S' \models \varphi'$.*

Proof Sketch. Consider $b = \max_{s \in S, a \in Act} |t_E(s, a)|$. We define the components of S' such that $|t'_E(s, a)| = b$ for all $s \in S'$, $a \in Act$, and $S \models \varphi$ iff $S' \models \varphi'$. S' and I' are defined by $S' = S \times \{0, 1\}$ and $I' = I \times \{1\}$. Intuitively, the added dimension is used to indicate whether a state is valid (1) or not (0). The agent's protocol function $p'((s, f))$ is defined as $p'((s, f)) = p(s)$ for each $s \in S$, $f \in \{0, 1\}$. The transition function $t'_E((s, f), a)$ returns $t_E(s, a) \times \{1\} \cup \{(s_1, 0), \dots, (s_{b-l}, 0)\}$ where $|t_E(s, a)| = l$ and $s_1, \dots, s_{b-l}$ are pairwise distinct states from S . The formula φ' adds $(m + 1) = 1$ conjunctively to each atom in φ , where m is such that $S = \mathbb{R}^m$. $\square$

In the rest of this section we assume that $|t_E(s, a)| = k$ for all s and a , and that t_E is given as k Alinearly definable Aunctions $t_i: \mathbb{R}^{m+n} \rightarrow \mathbb{R}^m$.

In what follows, we give the MILP encoding of the reachability problem, an extension of the method presented in [2].

As a stepping stone, we first encode the computation of a next environment state as a composition of the protocol function prot and of the transition functions t_i . Since, prot and, by assumption, each t_i is a piecewise-linear function, the function $\mathbf{y} = t_i(\mathbf{x}, \text{prot}(\mathbf{x}))$ is expressible as a set of MILP constraints denoted by $\Pi_i(\mathbf{x}, \mathbf{y})$. Moreover, solutions of $\Pi_i(\mathbf{x}, \mathbf{y})$ represent pairs of consecutive environment states:

Lemma 3.2. *Let $\Pi_i(\mathbf{x}, \mathbf{y})$ be an MILP program corresponding to $\mathbf{y} = t_i(\mathbf{x}, \text{prot}(\mathbf{x}))$. Given two states s and s' in $\mathcal{M}_S$, we have that $s' = t_i(s, \text{prot}(s))$ iff there is an assignment a to $\text{vars}(\Pi_i(\mathbf{x}, \mathbf{y}))$ such that $s = a(\mathbf{x})$, $s' = a(\mathbf{y})$, and $a \models \Pi_i(\mathbf{x}, \mathbf{y})$.*

Following Lemma 3.2, given a LE-CPS $\mathcal{S}$ and a bounded CTL formula φ , we construct a MILP program $\Pi_{\mathcal{S}, \varphi}(\mathbf{x})$, whose feasibility corresponds to the existence of a state in $\mathcal{M}_S$ 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. Also, given a binary variable δ and a set of constraints π , we write $(\delta = i) \Rightarrow \pi$ for the set of constraints $\{(\delta = i) \Rightarrow c \mid c \in \pi\}$.

We now define the MILP program $\Pi_{\mathcal{S}, \varphi}$.

Definition 3.1. *Given a LE-CPS $\mathcal{S}$ and a bounded CTL formula φ , their associated monolithic MILP encoding $\Pi_{\mathcal{S}, \varphi}$ is defined as the MILP program $\Pi_{\mathcal{S}, \varphi}(\mathbf{x})$, where $\mathbf{x}$ is a tuple of fresh state variables, and $\Pi_{\mathcal{S}, \varphi}(\mathbf{x})$ is built inductively using the rules:*

$$\begin{aligned} \Pi_{\mathcal{S}, \alpha}(\mathbf{x}) &= \{\Pi_{\alpha}(\mathbf{x})\} \\ \Pi_{\mathcal{S}, \varphi_1 \vee \varphi_2}(\mathbf{x}) &= (\delta = 1) \Rightarrow (\Pi_{\mathcal{S}, \varphi_1}(\mathbf{x}_1) \cup \{\mathbf{x} = \mathbf{x}_1\}) \cup \\ &\quad (\delta = 0) \Rightarrow (\Pi_{\mathcal{S}, \varphi_2}(\mathbf{x}_2) \cup \{\mathbf{x} = \mathbf{x}_2\}) \\ \Pi_{\mathcal{S}, \varphi_1 \wedge \varphi_2}(\mathbf{x}) &= \Pi_{\mathcal{S}, \varphi_1}(\mathbf{x}) \cup \Pi_{\mathcal{S}, \varphi_2}(\mathbf{x}) \\ \Pi_{\mathcal{S}, EX\varphi}(\mathbf{x}) &= \bigcup_{i=1}^k (\delta_i = 1) \Rightarrow (\Pi_i(\mathbf{x}, \mathbf{y}_i) \cup \{\mathbf{y} = \mathbf{y}_i\}) \\ &\quad \cup \{\delta_1 + \dots + \delta_k = 1\} \cup \Pi_{\mathcal{S}, \varphi}(\mathbf{y}) \\ \Pi_{\mathcal{S}, AX\varphi}(\mathbf{x}) &= \Pi_1(\mathbf{x}, \mathbf{y}_1) \cup \dots \cup \Pi_k(\mathbf{x}, \mathbf{y}_k) \cup \\ &\quad \Pi_{\mathcal{S}, \varphi}(\mathbf{y}_1) \cup \dots \cup \Pi_{\mathcal{S}, \varphi}(\mathbf{y}_k), \end{aligned}$$

where $\Pi_{\alpha}(\mathbf{x})$ is the linear constraint corresponding to the atomic proposition α and defined over $\mathbf{x}$, and in each inductive case, the binary variables $\delta, \delta_1, \dots, \delta_k$, the state variables $\mathbf{y}_1, \dots, \mathbf{y}_k, \mathbf{y}$, and all remaining auxiliary variables in $\Pi_i(\mathbf{x}, \mathbf{y}_i)$ are fresh (not used anywhere else).

Note that in general, $\Pi_{\mathcal{S}, \varphi}$ is exponentially large in the temporal depth of φ and polynomially large in the size of $\mathcal{S}$. We can prove that $\Pi_{\mathcal{S}, \varphi}$ is as intended.

Lemma 3.3. *Given a LE-CPS $\mathcal{S}$ and a bounded CTL formula φ , the following holds: there exists a state s in $\mathcal{M}_S$ such that $s \models \varphi$ iff there is an assignment a to $\text{vars}(\Pi_{\mathcal{S}, \varphi}(\mathbf{x}))$ such that $s = a(\mathbf{x})$ and $a \models \Pi_{\mathcal{S}, \varphi}(\mathbf{x})$.*

We can exploit Lemma 3.3 to define a procedure that solves the reachability problem by restricting $\mathbf{x}$ to only correspond to the initial states of $\mathcal{S}$. The procedure is given by Algorithm 1. Its soundness and completeness is shown by the following.

Theorem 3.1. *Given a LE-CPS $\mathcal{S} = (Ag, E, I)$ and a bounded CTL formula φ , Algorithm 1 returns True iff there exists $i \in I$ such that $\mathcal{A} \models \varphi$.*

Algorithm 1 The reachability procedure.

```

1: procedure REACHABLE( $\mathcal{S}, \varphi$ )
2:   Input: LE-CPS  $\mathcal{S} = (Ag, E, I)$ ; bounded CTL formula  $\varphi$ 
3:   Output: True/False
4:    $\varphi_I \leftarrow$  Boolean formula representing  $I$ 
5:    $\Pi_{\mathcal{S}, \varphi \wedge \varphi_I} \leftarrow$  MILP associated with  $\mathcal{S}$  and  $\varphi \wedge \varphi_I$ 
6:    $feasible \leftarrow$  MILP_SOLVER( $\Pi_{\mathcal{S}, \varphi \wedge \varphi_I}$ )
7:   return  $feasible$ 

```

We conclude by observing that the restriction to the bounded fragment of CTL is justified by the fact that checking reachability against unbounded CTL is undecidable, even in the case of the environment with deterministic transition function.

4 Reachability analysis for recurrent LE-CPSs

Finally, we show how to tackle the reachability problem in the case of recurrent LE-CPSs where in contrast to feed-forward LE-CPSs, the LEC is stateful and is implemented by an RNN R .

To provide a method for reachability analysis of RNN-LE-CPS, we rely on unrolling the RNN R into a feed-forward network F_d for a temporal depth d [1]. The verification procedure is then similar to the one described in Section 3. Formally, consider an RNN-LE-CPS $\mathcal{S} = ((Act, \text{prot}), (S, t_E), I)$, where $\text{prot} : (\mathbb{R}^m)^* \rightarrow \mathbb{R}^n$ is the function computed by an RNN R . Given an unrolling F_d of R , denote by $\text{prot}_d : (\mathbb{R}^m)^d \rightarrow \mathbb{R}^n$ the function computed by F_d . As before, assume that $|t_E(s, a)| = k$ for all s and a , and that t_E is given as k linearly definable functions $t_i : \mathbb{R}^{m+n} \rightarrow \mathbb{R}^m$.

We define recursively a MILP program $\Pi_{\mathcal{S}, \varphi}^d(\{\mathbf{x}_0, \dots, \mathbf{x}_d\})$ defined on an explicit temporal depth parameter d and a set of tuples of variables referring to consecutive environment states required as input to the unrolling of the RNN. Feasibility of this program corresponds to the existence of a sequence $s_0, \dots, s_d$ of states in $\mathcal{M}_{\mathcal{S}}$ such that s_d satisfies φ .

Definition 4.1. Given an RNN-LE-CPS $\mathcal{S}$ and a bounded CTL formula φ , we associate with them a MILP program $\Pi_{\mathcal{S}, \varphi}$ set to be $\Pi_{\mathcal{S}, \varphi}^0(\{\mathbf{x}_0\})$, which is recursively defined as:

$$\begin{aligned}
\Pi_{\mathcal{S}, \alpha}^d(\{\mathbf{x}_0, \dots, \mathbf{x}_d\}) &= \{\Pi_{\alpha}(\mathbf{x}_d)\} \\
\Pi_{\mathcal{S}, \varphi_1 \vee \varphi_2}^d(\{\mathbf{x}_0, \dots, \mathbf{x}_d\}) &= (\delta = 1) \Rightarrow (\Pi_{\mathcal{S}, \varphi_1}^d(\{\mathbf{x}_0^1, \dots, \mathbf{x}_d^1\}) \cup \bigcup \{\mathbf{x}_j = \mathbf{x}_j^1\}) \cup \\
&\quad (\delta = 0) \Rightarrow (\Pi_{\mathcal{S}, \varphi_2}^d(\{\mathbf{x}_0^2, \dots, \mathbf{x}_d^2\}) \cup \bigcup \{\mathbf{x}_j = \mathbf{x}_j^2\}) \\
\Pi_{\mathcal{S}, \varphi_1 \wedge \varphi_2}^d(\{\mathbf{x}_0, \dots, \mathbf{x}_d\}) &= \Pi_{\mathcal{S}, \varphi_1}^d(\{\mathbf{x}_0, \dots, \mathbf{x}_d\}) \cup \Pi_{\mathcal{S}, \varphi_2}^d(\{\mathbf{x}_0, \dots, \mathbf{x}_d\}) \\
\Pi_{\mathcal{S}, EX\varphi}^d(\{\mathbf{x}_0, \dots, \mathbf{x}_d\}) &= \bigcup_{i=1}^k (\delta_i = 1) \Rightarrow (\Pi_i^d(\mathbf{x}_0, \dots, \mathbf{x}_d, \mathbf{y}_i) \cup \{\mathbf{y} = \mathbf{y}_i\}) \\
&\quad \cup \{\delta_1 + \dots + \delta_k = 1\} \cup \Pi_{\mathcal{S}, \varphi}^{d+1}(\{\mathbf{x}_0, \dots, \mathbf{x}_d, \mathbf{y}\}) \\
\Pi_{\mathcal{S}, AX\varphi}^d(\{\mathbf{x}_0, \dots, \mathbf{x}_d\}) &= \Pi_1^d(\mathbf{x}_0, \dots, \mathbf{x}_d, \mathbf{y}_1) \cup \dots \cup \Pi_k^d(\mathbf{x}_0, \dots, \mathbf{x}_d, \mathbf{y}_k) \cup \\
&\quad \Pi_{\mathcal{S}, \varphi}^{d+1}(\{\mathbf{x}_0, \dots, \mathbf{x}_d, \mathbf{y}_1\}) \cup \dots \cup \Pi_{\mathcal{S}, \varphi}^{d+1}(\{\mathbf{x}_0, \dots, \mathbf{x}_d, \mathbf{y}_k\}),
\end{aligned}$$

where $d \geq 0$, Π_{α} is defined as before, and $\Pi_i^d(\mathbf{x}_0, \dots, \mathbf{x}_d, \mathbf{y})$ is the set of MILP constraints corresponding to $\mathbf{y} = t_i(\mathbf{x}_d, \text{prot}_d(\mathbf{x}_0 \cup \dots \cup \mathbf{x}_d))$.

Having defined $\Pi_{\mathcal{S}, \varphi}$, the reachability procedure in the case of RNN-LE-CPSs is exactly the same as the case of feed-forward LE-CPSs. Observe that since we check reachability against

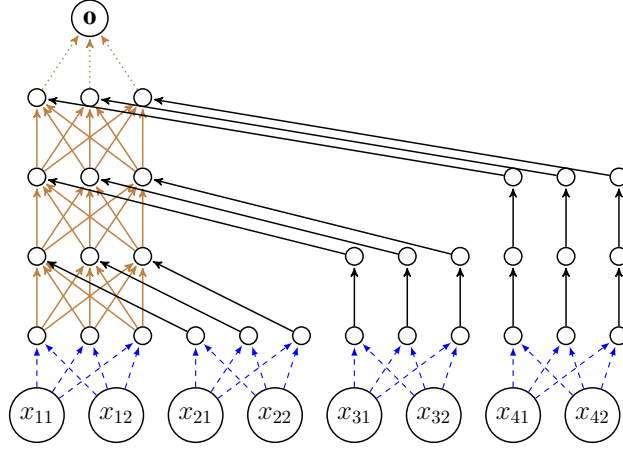


Figure 1: Unrolling using Input on Start

bounded specifications, the method is sound and complete despite the fact that the feed-forward unrollings are only abstractions of the recurrent networks.

We conclude by describing two methods of unrolling the RNN: Input on Start and Input on Demand [1].

Input on Start (IOS). In this method, all the input values are initially scaled according to the weights of the $W_{(i \rightarrow h)}$ matrix. Then, at the time step when the input is needed it is passed unchanged to the corresponding hidden unit.

Before defining this construction, we introduce some notation. We denote by I_a the $a \times a$ identity matrix, by $O_{a,b}$ the $a \times b$ zero matrix, and by $A \otimes B$ is the Kronecker product [3] of matrices A and B defined by:

$$(A \otimes B)_{i,j} = A_{\lfloor (i-1)/p \rfloor + 1, \lfloor (j-1)/q \rfloor + 1} \cdot B_{(i-1)\%p+1, (j-1)\%q+1},$$

where $\lfloor a \rfloor$ defines the floor of a and $a\%b$ represents the remainder of a/b .

Now, we can proceed to define the construction of our unrolling. This is shown visually in Figure 1. We formalise the construction in the definition below.

Definition 4.2. Let R be an RNN with weight matrices $W_{(i \rightarrow h)}$, $W_{(h \rightarrow h)}$ and $W_{(h \rightarrow o)}$. Let $d \in \mathbb{Z}^+$ be the length of a sequence of inputs. Then, the IOS d -unrolling of R , denoted F_d^{ios} , is the FFNN with d hidden layers defined by assigning to each layer the weights:

$$\begin{aligned} W_1 &= I_s \otimes W_{(i \rightarrow h)}, \\ W_l &= \begin{bmatrix} W_{(h \rightarrow h)} & O_{|H|, (s-l) \cdot |H|} \\ I_{(s-l)} \otimes I_{|H|} & \end{bmatrix} \text{ for } l \in \{2, \dots, d\}, \\ W_{d+1} &= W_{(h \rightarrow o)}. \end{aligned}$$

Notice that the input layer of the network applies the transformation defined by $W_{(i \rightarrow h)}$ to the input data. The hidden layers each take in another element of the transformed input data and combine it with the existing state using the weights in $W_{(h \rightarrow h)}$. Finally, the output layer applies the transformation given by $W_{(h \rightarrow o)}$.

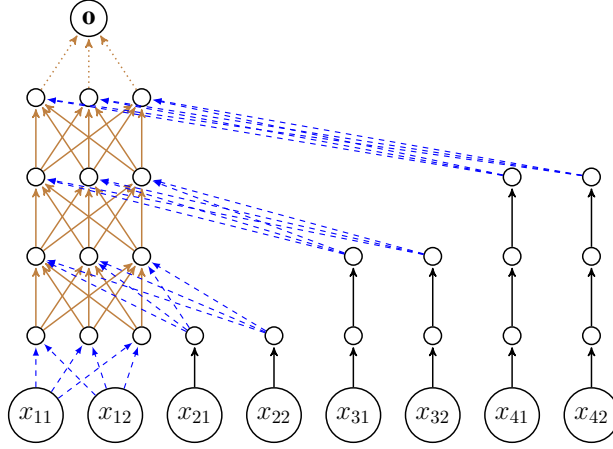


Figure 2: Unrolling using Input on Demand

We now show that given an RNN R , the FFNN F_d^{ios} constructed by following Definition 4.2 is an appropriate abstraction. Specifically, the outputs produced by F_d^{ios} are the same as those produced by R on any input sequence of length d .

Theorem 4.1. *Let $f : (\mathbb{R}^x)^* \rightarrow \mathbb{R}^y$ denote the function computed by R and $g : \mathbb{R}^{x^d} \rightarrow \mathbb{R}^y$ denote the function computed by F_d^{ios} . Then, for all $\mathbf{x} \in (\mathbb{R}^x)^d$, it is the case that $f(\mathbf{x}) = g((\mathbf{x}_{0,0}, \dots, \mathbf{x}_{x,d}))$.*

Proof sketch. The proof relies on showing that the output of layer k of the FFNN for $k < d$ is equal to the value of h_k^x in the definition of recurrent networks in WP1.1 M12 Deliverable. Then, the output of layer $d + 1$ corresponds to the output of the RNN. $\square$

Input on Demand (IOD). This unrolling method differs from the IOS method by passing on the input unchanged to the hidden units, rather than transforming the input at the start. This is illustrated in Figure 2.

Definition 4.3. *Let R be an RNN with weight matrices $W_{(i \rightarrow h)}$, $W_{(h \rightarrow h)}$ and $W_{(h \rightarrow o)}$. Let $n \in \mathbb{Z}^+$ be the length of sequences that we wish to consider. Then, the IOD d -unrolling of R , denoted F_d^{iod} , is the FFNN with d hidden layers defined by assigning to each layer the weights:*

$$\begin{aligned}
 W_1 &= \begin{bmatrix} W_{(i \rightarrow h)} & O_{|I|, |I|(s-1)} \\ O_{(|I|(s-1)), |H|} & I_{|I|(s-1)} \end{bmatrix}, \\
 W_l &= \begin{bmatrix} W_{(h \rightarrow h)} & O_{|H|, |I|(s-l-1)} \\ W_{(i \rightarrow h)} & O_{|I|, |I|(s-l-1)} \\ O_{(|I|(s-l-1)), |H|} & I_{|I|(s-l-1)} \end{bmatrix} \text{ for } l \in \{2, \dots, d\}, \\
 W_{d+1} &= W_{(h \rightarrow o)}.
 \end{aligned}$$

Notice that in this case the input layer applies the transformation given by $W_{(i \rightarrow h)}$ to the first element of the sequence, and passes the rest of the sequence along unchanged. Each hidden layer

then applies $W_{(i \rightarrow h)}$ to another element of the sequence and combines this with the existing state using $W_{(h \rightarrow h)}$. Finally, the output layer applies the transformation given by $W_{(h \rightarrow o)}$.

As for IOS, we now show that the IOD construction also produces an appropriate abstraction which is behaviourally equivalent to the original RNN for input sequences of length n .

Theorem 4.2. *Let $f : (\mathbb{R}^x)^n \rightarrow \mathbb{R}^y$ denote the function computed by R and $g : \mathbb{R}^{xn} \rightarrow \mathbb{R}^y$ denote the function computed by R'_n . Then, for all $\mathbf{x} \in (\mathbb{R}^x)^n$, it is the case that $f(\mathbf{x}) = g(\mathbf{x}_{0,0}, \dots, \mathbf{x}_{x,n})$.*

Proof. The proof is similar to that of Theorem 4.1. □

5 Conclusions

In this deliverable we put forward methods for reachability analysis of LE-CPSs against specification properties from a bounded fragment of CTL and of single LECs against Boolean combinations of linear constraints on their outputs. These methods are sound and complete and are based on recasting the reachability problem as the feasibility problem of a mixed-integer linear program (MILP). When the analysis returns a positive answer, it is possible to retrieve the inputs for which the specification is satisfied. In the case when it is an undesirable property/output was shown to be reachable, these inputs can be used to debug the LE-CPS or the LEC at hand.

The methods developed here are at the basis of the procedures implemented in the Venus verification toolkit.

Note that the presented algorithm has been designed with correctness, but not with efficiency in mind, and so a naive implementation is not going to be feasible for large neural networks. Therefore, the main directions for future work are concerned with scalability. It includes work on devising optimisation techniques for the above MILP-based method, but also possibly on combining it with fast incomplete approaches.

References

- [1] M. E. Akintunde, A. Kevorchian, A. Lomuscio, and E. Pirovano. Verification of RNN-based neural agent-environment systems. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI19)*. AAAI Press, 2019. To Appear.
- [2] M. E. Akintunde, A. Lomuscio, L. Maganti, and E. Pirovano. Reachability analysis for neural agent-environment systems. In *Proceedings of the 16th International Conference on Principles of Knowledge Representation and Reasoning (KR18)*, pages 184–193. AAAI Press, 2018.
- [3] J. Brewer. Kronecker products and matrix calculus in system theory. *IEEE Transactions on Circuits and Systems*, 25(9):772–781, 1978.
- [4] R. .R. Bunel, I. Turkaslan, P. Torr, P. Kohli, and P. K. Mudigonda. A unified view of piecewise linear neural network verification. In *Proceedings of the 31st Annual Conference on Neural Information Processing Systems (NeurIPS18)*, pages 4790–4799. Curran Associates, Inc., 2018.
- [5] I. Griva, S. Nash, and A. Sofer. *Linear and nonlinear optimization*, volume 108. Siam, 2009.

- [6] A. Lomuscio and L. Maganti. An approach to reachability analysis for feed-forward relu neural networks. *CoRR*, abs/1706.07351, 2017.

Low foot-print methods for reachability analysis for feed-forward ReLU LE-CPSs

Imperial College London

1 Introduction

This deliverable reports low foot-print methods for reachability analysis for feed-forward ReLU LE-CPSs. In this section we recall some definitions and notions we exploit in the later sections.

We start by reporting advancements in dealing with the following two reachability problems

- Given a neural network N computing function f_N , a set I of initial values and a condition C on the output of N , decide whether there exists $i \in I$ such that $C(f_N(i))$ holds.
- Given an LE-CPS $\mathcal{S}$ with non-deterministic environment, a set I of initial states, and a bounded CTL specification φ , decide whether there exists an initial state $s_0 \in I$ such that $\mathcal{M}_{\mathcal{S}}, s_0 \models \varphi$.

Exact formulations of the above problems can be given as mixed integer linear programs: the corresponding MILP program is feasible if and only if the reachability problem is satisfied. Therefore our main objective here is to produce more optimal MILP formulations.

In Section 2 we report a novel verification procedure for the former reachability problem that relies on *dependency analysis* to reduce the search space that needs to be explored in order to find a feasible assignment, published at AAAI 2020. We show empirically that combined with other techniques from the literature dependency analysis generates speed-ups of at least one order of magnitude against state-of-the-art tools.

In Section 3 we present a compositional verification procedure for the latter reachability problem that allows for parallel processing and in particular very efficient at finding witnesses, published at AAMAS 2020. We evaluate the compositional and the monolithic procedures on two use-cases, and the experiments confirm that the compositional procedure is very efficient at finding witnesses, while the monolithic procedure can be more efficient at proving safety (when unsafe states are not reachable).

Then in Section 4 present a method for reachability analysis for LE-CPS consisting of multiple LECs. In this case we are interested in reasoning about strategic abilities of LE-CPS components and coalitions thereof, so we consider bounded ATL specifications (defined in WP1.1 M18 Deliverable). We define the semantics of such LE-CPSs, which we call *neural interpreted systems*, and devise an MILP-based verification procedure against bounded ATL specifications. Preliminary experiments show that more powerful methods for dealing with such complex systems are required.

2 Fast Reachability Analysis using Dependencies

This section reports the results of the paper “Efficient Verification of ReLU-based Neural Networks via Dependency Analysis” published at AAAI 2020. Namely, we introduce an efficient method for the verification of ReLU-based feed-forward neural networks. We derive an automated procedure that exploits dependency relations between the ReLU nodes, thereby pruning the search tree that needs to be considered by MILP-based formulations of the verification problem. We augment the resulting algorithm with methods for input domain splitting and symbolic interval propagation. We evaluate our toolkit *Venus* on the ACAS collision avoidance networks and models trained on the MNIST and CIFAR-10 datasets. The experimental results obtained indicate considerable gains over the present state-of-the-art tools.

2.1 Preliminaries

For the sake of self-containedness and fixing the notation, we present the basic notions and definitions relied on later on in the section.

Feed-forward neural networks. A feed-forward neural network (FFNN) is a directed acyclic graph whose nodes are structured in layers. The first layer is the *input layer*, also referred to as layer 0, the last layer is the *output layer*, also referred to as layer k , and every layer in-between is a *hidden layer*, also referred to as layer i , for $1 \leq i < k$. Every node other than an input node is connected to every node in the preceding layer. Each edge is associated with a weight, which is learned during the training phase. Given an input vector, the network computes a function by propagating the input through the network, where, at each step, a node’s *output* results from applying an activation function to the *pre-activation* of the node, which is the weighted sum of the outputs of the nodes from the previous layer. Here, we only consider the *ReLU* activation function defined by $\text{ReLU}(x) \triangleq \max(x, 0)$ for $x \in \mathbb{R}$.

We denote by s_i the number of nodes in layer i . We use $n_{i,q}$ to refer to the q -th node of layer i . Given an input $\mathbf{x}$ to the network, we write $\hat{\mathbf{x}}_{i,q}$ ($\mathbf{x}_{i,q}$, respectively) for the pre-activation (output, respectively) of the node $n_{i,q}$. For a set of inputs over a bounded domain, every node is associated with lower and upper pre-activation and activation bounds. We write $\hat{\mathbf{l}}_{i,q}$ and $\hat{\mathbf{u}}_{i,q}$ ($\mathbf{l}_{i,q}$ and $\mathbf{u}_{i,q}$, respectively) for the pre-activation’s (output’s, respectively) lower and upper bounds. Similarly, $\hat{\mathbf{x}}_i$ and $\mathbf{x}_i$ refer to the vector of pre-activations and outputs of layer i over domains $[\hat{\mathbf{l}}_i, \hat{\mathbf{u}}_i]$ and $[\mathbf{l}_i, \mathbf{u}_i]$, respectively, where $\mathbf{x}_0 = \mathbf{x}$, and $\mathbf{l}_0$ and $\mathbf{u}_0$ are the input lower and upper bounds. We write W_i and b_i to refer to the weight matrix and bias vector of layer i , $i \geq 1$, respectively.

Given an input $\mathbf{x}$ and for $i \geq 1$, the output $\mathbf{x}_i$ of layer i is computed from $\mathbf{x}_{i-1}$ by applying the function $f_i: \mathbb{R}^{s_{i-1}} \rightarrow \mathbb{R}^{s_i}$, which is defined as $f_i(\mathbf{x}_{i-1}) \triangleq \text{ReLU}(W_i \mathbf{x}_{i-1} + b_i) = \text{ReLU}(\hat{\mathbf{x}}_i)$, where the ReLU function is applied element-wise. Given the above, a neural network of $k+1$ layers, is defined as a function $f: \mathbb{R}^{s_0} \rightarrow \mathbb{R}^{s_k}$, corresponding to the composition of the functions f_i computed by each layer i , i.e., $f(\mathbf{x}) \triangleq f_k(\dots f_1(\mathbf{x}) \dots)$.

A ReLU node $n_{i,q}$ can be in one of two states. It is in the *strictly active* state (or, is strictly active), denoted $\text{st}(n_{i,q}) = \top$ if $\hat{\mathbf{l}}_{i,q} \geq 0$. It is in the *strictly inactive* state (or, is strictly inactive), denoted $\text{st}(n_{i,q}) = \perp$, if $\hat{\mathbf{u}}_{i,q} \leq 0$. A *stable* node is a node that is either strictly active or strictly inactive. Otherwise, the node is said to be *unstable*, denoted $\text{st}(n_{i,q}) = ?$.

MILP formulation. The reachability problem admits a *precise* representation as a Mixed Integer Linear Program (MILP) by means of the “big-M” encoding [3]. Specifically, the corresponding MILP program is feasible iff the answer to the verification problem is *yes*. Assuming the pre-activation bounds of the nodes have already been calculated (see below), the MILP encoding of a node $n_{i,q}$ depends on its state. If the node is strictly active, then it can be encoded

by $\mathbf{x}_{i,q} = \hat{\mathbf{x}}_{i,q}$. If the node is strictly inactive, then it can be encoded by $\mathbf{x}_{i,q} = 0$. Otherwise, the encoding of the node is given by:

$$\begin{aligned} \mathbf{x}_{i,q} &\geq 0, & \mathbf{x}_{i,q} &\geq \hat{\mathbf{x}}_{i,q}, \\ \mathbf{x}_{i,q} &\leq \hat{\mathbf{u}}_{i,q} \cdot \delta_{i,q}, & \mathbf{x}_{i,q} &\leq \hat{\mathbf{x}}_{i,q} - \hat{\mathbf{1}}_{i,q} \cdot (1 - \delta_{i,q}), \end{aligned}$$

where $\delta_{i,q}$ is a binary variable such that $\delta_{i,q} = 0$ iff $\mathbf{x}_{i,q} = 0$ and $\delta_{i,q} = 1$ iff $\mathbf{x}_{i,q} = \hat{\mathbf{x}}_{i,q}$.

For an MILP program comprising a set Δ of binary variables, a (partial) *configuration* is a (partial) function $h : \Delta \rightarrow \{0, 1\}$ that assigns to each variable (some of the variables) a value from $\{0, 1\}$. The set of all possible partial configurations is said to be the program's *configuration space*.

A leading approach for solving MILP programs is the branch-and-bound method. In branch-and-bound, each integrality constraint $\delta_{i,j} \in \{0, 1\}$ is relaxed to a linear constraint $\delta_{i,j} \in [0, 1]$, thereby defining a linear program which can be solved in polynomial-time [17]. Integrality is iteratively enforced by dividing the search domain into sub-regions excluding fractional solutions. In the context of neural networks, the efficacy of branch-and-bound depends on (i) the number of binary variables, i.e., the number of unstable nodes, and (ii) the tightness of the linear relaxations, i.e., the tightness of the pre-activation bounds.

2.2 Dependency Analysis

A major impediment to the scalable verification of ReLU-based FFNNs is the configuration space generated by the piecewise linearity of the ReLU nodes. Several approaches have been put forward for reducing the number of non-linearities that need to be considered for solving the verification problem. In particular, techniques that split the input domain have been shown effective in stabilising the ReLU nodes, thereby generating easier verification problems whose solutions can be combined to decide the original problem in a more efficient manner. Still, since the number of subproblems that need to be processed grows exponentially in the number of splits, networks with high input dimensionality remain hard to tackle. To overcome this, we introduce a technique that exploits what we define below as the *network's dependency relation* to reduce the configuration space that needs to be considered in solving a verification problem. We then show how to detect such dependency relations and utilise them during MILP-based verification.

Informally, the network's dependency relation can be used to stabilise a ReLU node on the basis of an assumed stable state of another node. Formally it is defined as follows.

Definition 1 (Dependency relation). *Given a neural network f that comprises a set of unstable nodes U , the dependency relation for U , $\mathcal{D}_f \subseteq U \times U$ is the set of all pairs $(n_{i,q}, n_{j,r})$ such that $\text{st}(n_{i,q}) \neq ? \implies \text{st}(n_{j,r}) \neq ?$.*

A node $n_{j,r}$ depends on a node $n_{i,q}$ if whenever $n_{i,q}$ is either strictly active or inactive, then $n_{j,r}$ has to be either strictly active or inactive. It follows that the configuration space generated by a branch-and-bound method can be reduced by stabilising $n_{j,r}$ whenever $n_{i,q}$ becomes stable. In particular, for a network with n unstable nodes, there are 2^{n-2} configurations that violate a given dependency; therefore, each dependency provides a means to reduce the configuration space by a factor of $1/4$.

Example 1. *Consider the network shown in the left part of Figure 1. In the figure each interval next to a node denotes the pre-activation bounds of the node. Note that nodes $n_{1,2}$ and $n_{2,1}$ are unstable. Assume that a branch-and-bound method branches on node $n_{1,2}$, thereby splitting the optimisation problem into two sub-problems: one where $n_{1,2}$ is strictly active and one where $n_{1,2}$*

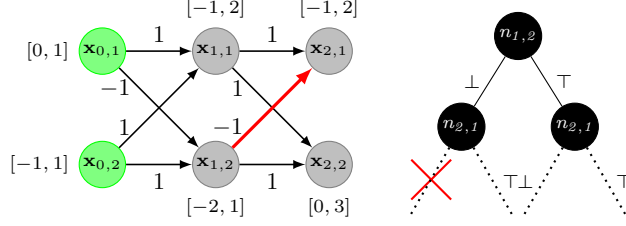


Figure 1: **Left:** Feedforward neural network exhibiting the dependency “if $n_{1,2}$ is inactive, then $n_{2,1}$ is inactive”. **Right:** Depiction of the configuration space reduction induced by the dependency.

is strictly inactive. Consider the latter sub-problem. We have that $\mathbf{l}_{1,2} = 0$ and $\mathbf{u}_{1,2} = 0$. Therefore, $\hat{\mathbf{l}}_{2,1} = 1 \cdot 0 + -1 \cdot 0 = 0$ and $\hat{\mathbf{u}}_{2,1} = 1 \cdot 2 - 1 \cdot 0 = 2$. Hence, $n_{2,1}$ is strictly active, and consequently, $(n_{i,j}, n_{q,r}) \in \mathcal{D}_f$. The right part of Figure 1 depicts the configuration space satisfying said dependency.

We now proceed to derive a procedure for computing a network’s dependency relations. To ease the presentation, we express dependency relations as unions of four disjoint sets $\mathcal{D}_f = \bigcup_{z, z' \in \{\top, \perp\}} \mathcal{D}_f^{z, z'}$, where each $\mathcal{D}_f^{z, z'}$ comprises dependencies pertaining to the ReLU states z and z' , i.e., $\mathcal{D}_f^{z, z'} \triangleq \{(n_{i,q}, n_{j,r}) \mid \text{st}(n_{i,q}) = z \implies \text{st}(n_{j,r}) = z'\}$. Also, we distinguish between inter- and intra-layer dependencies, which require a different algorithmic treatment. We begin by studying dependencies in the same layer.

Intra-layer dependencies. A dependency $(n_{i,q}, n_{j,r})$ is said to be an *intra-layer dependency* if $i = j$. To compute the dependency relation, given a pair of nodes, we compute the lower and upper bounds of a node under the assumption that the pre-activation of the other is zero, and use the bounds to determine the dependencies.

Formally, for a pair of nodes $n_{i,q}, n_{i,r}$, we define $\hat{\mathbf{x}}_{i,q,r=0}$ as the set of pre-activations of $n_{i,q}$ when the pre-activation of $n_{i,r}$ is zero:

$$\hat{\mathbf{x}}_{i,q,r=0} \triangleq \{(W_i)_q \cdot \mathbf{x}_{i-1} + (b_i)_q \mid (W_i)_r \mathbf{x}_{i-1} + (b_i)_r = 0\}.$$

Geometrically, this can be viewed as the intersection of the plane generated by the pre-activations of $n_{i,q}$ and $n_{i,r}$ with $\hat{\mathbf{x}}_{i,r} = 0$. Note that said intersection always exists as both $n_{i,q}$ and $n_{i,r}$ are unstable; therefore, there is an input for which their pre-activations equal zero. On the basis of the lower and upper bounds of $\hat{\mathbf{x}}_{i,q,r=0}$ and $\hat{\mathbf{x}}_{i,r,q=0}$, which can be computed as standard using interval arithmetic, the following lemma identifies the intra-layer dependencies (see Figure 2).

Lemma 1. For a neural network f and a pair of unstable nodes $(n_{i,q}, n_{i,r})$, the following hold:

1. $(n_{i,q}, n_{i,r}) \in \mathcal{D}_f^{\top, \perp}$ iff $\hat{\mathbf{u}}_{i,q,r=0} < 0$ and $\hat{\mathbf{u}}_{i,r,q=0} < 0$.
2. $(n_{i,q}, n_{i,r}) \in \mathcal{D}_f^{\perp, \top}$ iff $\hat{\mathbf{l}}_{i,q,r=0} > 0$ and $\hat{\mathbf{l}}_{i,r,q=0} > 0$.
3. $(n_{i,q}, n_{i,r}) \in \mathcal{D}_f^{\top, \top}$ iff $\hat{\mathbf{u}}_{i,q,r=0} < 0$ and $\hat{\mathbf{l}}_{i,r,q=0} > 0$.
4. $(n_{i,q}, n_{i,r}) \in \mathcal{D}_f^{\perp, \perp}$ iff $\hat{\mathbf{l}}_{i,q,r=0} > 0$ and $\hat{\mathbf{u}}_{i,r,q=0} < 0$.

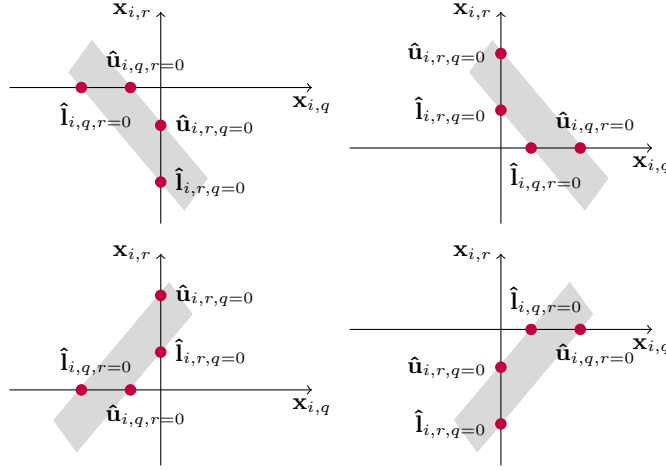


Figure 2: The types of intra-layer dependencies. **Top-left:** if $n_{i,q}$ is active, then $n_{i,r}$ is inactive. **Top-right:** if $n_{i,q}$ is inactive, then $n_{i,r}$ is active. **Bottom-left:** if $n_{i,q}$ is active, then $n_{i,r}$ is active. **Bottom-right:** If $n_{i,q}$ is inactive, then $n_{i,r}$ is inactive.

Lemma 1 gives a procedure for identifying intra-layer dependencies by computing the right hand side of each of the above clauses for every pair of unstable nodes in a layer. Dependencies between layers require a different treatment.

Consecutive-layer and inter-layer dependencies. A dependency $(n_{i,q}, n_{j,r})$ is said to be an *inter-layer dependency* if $j \neq i$. A special case of inter-layer dependencies are those defined by $j = i + 1$, which we call *consecutive-layer dependencies*. As we show below the latter are sufficient to obtain the smallest possible configuration space.

Lemma 2. *Let S be the subset of a given network's configuration space that satisfies all consecutive-layer dependencies. Then, every configuration in S satisfies any inter-layer dependency $(n_{i,q}, n_{j,r}) \in \mathcal{D}_f^{z,z'}$ with $j - i > 1$.*

It follows that consecutive-layer dependencies are sufficient to determine the minimum configuration space. To obtain a procedure for calculating consecutive-layer dependencies we introduce the result below.

Lemma 3. *For a neural network f and a pair of unstable nodes $n_{i,q}, n_{j,r}$, for $j = i + 1$, the following hold:*

1. $(n_{i,q}, n_{j,r}) \in \mathcal{D}_f^{\perp, \perp} \Leftrightarrow \hat{\mathbf{u}}_{j,r} - (W_j)_{r,q} \cdot \mathbf{u}_{i,q} \leq 0$.
2. $(n_{i,q}, n_{j,r}) \in \mathcal{D}_f^{\perp, \top} \Leftrightarrow \hat{\mathbf{l}}_{j,r} - (W_j)_{r,q} \cdot \mathbf{u}_{i,q} \geq 0$.
3. $\mathcal{D}_f^{\top, \perp} = \emptyset$
4. $\mathcal{D}_f^{\top, \top} = \emptyset$

Lemma 3 gives a procedure for identifying consecutive-layer dependencies by checking the right hand side of clauses (1) and (2) for every pair of unstable nodes in consecutive layers.

Dependency analyser. Given the above, we now put forward a procedure using the identification of dependencies to reduce the configuration space. The procedure runs in conjunction

Distribution (A) Approved for public release; distribution is unlimited.

Algorithm 1 The dependency analysis procedure.

```

1: procedure DEPENDENCY ANALYSIS(milp,  $h$ )
2:   Input: MILP milp, partial configuration  $h$ .
3:   for each  $h(\delta_{i,q}) = 0$  do
4:      $\mathbf{l}_{i,q} \leftarrow 0, \mathbf{u}_{i,q} \leftarrow 0$ 
5:   Compute remaining bounds (Section 2).
6:   Compute  $\mathcal{D}_f$  (Lemmas 3 and 1).
7:   Add  $\{cut_{d,h} \mid d \in \mathcal{D}_f\}$  to milp (Definition 2)

```

with a MILP solver, where it builds a new constraint for each dependency which it adds to the program being analysed by the solver. This is performed at runtime during the branch-and-bound procedure, as the computation of the dependencies is consistent with the current, partial configuration of ReLU nodes being considered by the MILP solver. This allows for the identification of dependencies whilst several nodes have already been stabilised, as opposed to offline methods where most nodes would typically be unstable, thereby hindering the existence of dependencies, as it is rarer for a node to cause a state change on another.

Consider a partial configuration h being considered by the MILP solver. To determine its validity, the solver either extends it to a complete one that satisfies all constraints or to a partial one that violates at least one of the constraints. The dependency analysis procedure put forward here reduces the number of extensions of h that need to be evaluated. Algorithm 1 enumerates the steps of the procedure. First, it stabilises the ReLU nodes as per h and re-computes the bounds for the ones being unstable under h . On the basis of the bounds, it determines the dependencies as per Lemmas 3 and 1. These are then expressed as constraints, referred to as *dependency cuts*, which are added at runtime to the MILP program. The dependency cuts are defined as follows.

Definition 2 (Dependency cuts). *For a partial configuration $h: \Delta \rightarrow [0, 1]$, the associated dependency cut $cut_{d,h}$ of a dependency $d \triangleq (n_{i,q}, n_{j,r}) \in \mathcal{D}_f^{z,z'}$ is a MILP constraint defined as follows:*

$$cut_{d,h} \triangleq \gamma_{j,r}(z') \leq \sum_{h(\delta)=0} \delta + \sum_{h(\delta)=1} 1 - \delta + \gamma_{i,q}(z),$$

where $\gamma_{i,q}(z)$ equals $\delta_{i,q}$ if $z = \perp$ and $1 - \delta_{i,q}$ if $z = \top$.

A dependency cut derived from a configuration h is satisfied by an extension of h iff the extended configuration satisfies the corresponding dependency; it follows that each dependency cut removes from the search space all configurations extending h that do not satisfy the dependency. Additionally, for any configuration that does not extend h , the cut is trivially satisfied, thereby not altering the search space for those configurations. The former is shown by clause (1) and the latter is proved by clause (2) of the following theorem.

Theorem 1. *Let h be a partial configuration and $d \in \mathcal{D}_f^{z,z'}$ a dependency. Then, the following hold:*

1. *For every h' with $h \subseteq h'$, $h' \models cut_{d,h}$ iff $h' \models d$.*
2. *For every h' with $h \not\subseteq h'$, $h' \models cut_{d,h}$.*

The above concludes the description of the dependency analysis procedure. The procedure runs in time $\mathcal{O}(k \cdot s^2)$, where k is the number of layers and s is the layers' maximal size. Clearly running this procedure has a cost. In the next section we will experimentally evaluate how

Distribution (A) Approved for public release; distribution is unlimited.

	n_s	t_{all}	t_{solved}	av_s	$\frac{t_{all}}{t_{Venus}}$
MNIST (100 verification queries)					
Venus	100	5,953.46	573.38	9.10	—
Marabou	0	86,400.00	—	—	14.51
Neurify	65	126,007.24	7.00	0.11	21.17
NSVerify	95	26,906.81	2,515.15	39.9	4.52
CIFAR-10 (100 verification queries)					
Venus	100	857.11	560.04	7.36	—
Marabou	0	86,400.00	—	—	100.80
Neurify	76	87,178.46	778.46	10.24	101.71
NSVerify	100	6,898.64	3,460.41	45.53	8.04
ACAS XU (172 verification queries)					
Venus	170	19,642.57	5,527.76	36.36	—
Marabou	156	140,916.96	75,747.78	498.3	7.17
Neurify	167	23,628.75	2,555.38	16.81	1.2
NSVerify	6	86,400.00	—	—	30.77

Table 1: Experimental results obtained when running Venus, Marabou, Neurify and NSVerify.

frequent these calls should be. Also, note that since our dependency framework is a function of the bounds of the ReLU nodes, the procedure can further be optimised by using domain splitting and symbolic interval propagation methods, since these lead to tighter intervals for the ReLU nodes.

In the next section we show that all these factors combined improve the scalability of formal verification of neural networks over the state-of-the-art.

2.3 Experiments

We implemented the dependency analysis procedure in our toolkit Venus [30], together with symbolic interval propagation for estimating the bounds and a domain splitting technique. In this section we evaluate Venus on a number of widely used benchmarks and compare it against the state-of-the-art neural-network verification engines.

For the comparisons we restrict our attention to *complete* methods; while these are often less scalable than incomplete ones, they provide full guarantees on the correctness of their outputs, which is a key objective here. At present, the leading complete verification tools are Marabou [19] and Neurify [31]. To assess the improvement of Venus over plain MILP-based verification, we additionally compare Venus against NSVerify [3]. We used the most commonly used benchmarks in the context of FFNNs verification:

ACAS Xu [14] comprises 45 ReLU-based FFNNs, which were developed as part of an airborne collision avoidance system to advise horizontal steering decisions for unmanned aircrafts. We considered the specifications reported in [18]. Each was tested on all 45 networks, thereby giving rise (for a total of 10 specifications) to 172 verification problems. For the experiments, Venus was run with 2 splitters, 4 workers, the stability-ratio cutoff set to 0.7, the depth discount set to 20 and with the dependency analyser turned off; Neurify was run with MAX THREAD set to 2; Marabou was run with the parameters reported in [19].

MNIST [22] is a dataset comprising images of hand-written digits 0-9, each formatted as a 28x28x1-pixel grayscale image. We used MNIST to train a FFNN with 2 hidden layers, each comprising 512 neurons. We verified the network against local robustness for a perturbation

radius of 0.05 on 100 randomly selected images. For the experiments, we ran **Venus** with 2 splitters, 2 workers, the stability-ratio set to 0.4, the depth discount set to 4, and the dependency analyser turned on; **Neurify** was run with `MAX_THREAD` set to 1; **Marabou** was run with the parameters reported in [19].

CIFAR-10 [21] is a dataset comprising images of objects from 10 different classes (airplanes, cars, birds, cats, etc.). Each image is formatted as a 32x32x3-pixel colour image. We used CIFAR-10 to train a FFNN with 3 hidden layers, the first comprising 1024 neurons, and the second and third comprising 512 neurons. We verified the network against local robustness for a perturbation radius of 0.01 on 100 randomly selected images. We ran all tools with the same parameters as for MNIST.

All experiments were carried out on an Intel Core i7-7700K (4 cores) equipped with 16GB RAM, running Ubuntu 18.04. Each verification query had a *local* timeout of 1 hour. The sum of verification queries associated with each benchmark had a *global* timeout of 24 hours. Table 1 reports the experimental results. For each of the tools and benchmarks, the table gives the number n_s of verification queries that were solved, the overall time t_{all} taken for all queries, the overall time t_{solved} and the average time av_s taken for the queries that three best performing tools were able to solve, and the ratio between overall time taken t_{all} by a tool over that of **Venus**, t_{all}^{Venus} .

The results obtained on MNIST show that **Venus** was the most performing of the toolkits, both in terms of the overall verification time and the number of verification queries solved. **Neurify** was not able to analyse 35 of the queries because of excessive memory consumption. For these cases, we considered **Neurify** as having timed out. **Marabou** did not solve any of the queries under local and global timeouts. **NSVerify** performed better than both **Neurify** and **Marabou**. **Venus** was found 4.52 times faster than **NSVerify** and 21 times faster than **Neurify**. For CIFAR-10 the difference between **Venus** and the other tools was greater, suggesting that the higher the dimensionality and complexity of the model, the bigger the difference.

Venus’s performance was also found superior on ACAS XU, both in terms of the overall verification time and the number of queries solved. **Neurify** was the fastest tool w.r.t. the number of queries that all the tools could solve. **Marabou** solved a comparable number of queries to **Venus** and **Neurify** but was slower than both of them. **NSVerify** solved only 6 queries within the local and global timeouts.

Figure 3 gives a graphical representation of the total number of verification queries that each tool could verify as a function of time. In summary, **Venus** solved most verification instances after approx. 15 secs. Also, to the best of our knowledge, **Venus** is the only tool that can seemingly analyse both low-dimensional and high-dimensional networks, outperforming the state-of-the-art tools for each class, often by more than one order of magnitude. The only aspect we found **Venus** to be less performing was counterexample generation where **Neurify** was the fastest tool.

The experiments also suggest that the verification of networks with low input-dimensionality is particularly amenable to domain splitting, as domain splitting techniques act as effective configuration-space minimisers. As a result, branch-and-bound methods that combine domain splitting are advantageous over ones that do not, as indicated by the outperformance of **Venus** over **NSVerify** on ACAS.

Differently, for high-dimensional input domains, the experiments suggest that domain splitting methods do not significantly reduce the configuration space, as indicated by the degraded performance of **Neurify** and **Marabou** on MNIST and CIFAR-10. In contrast, techniques that directly target the reduction of the configuration space exhibit high efficacy over high-dimensional inputs, as exemplified by **Venus** and **NSVerify**. **Venus** is more effective than **NSVerify** by considering FFNN-specific configuration-space reductions. This suggests that MILP solvers are not necessarily best used as black boxes, but application-specific considerations can help to improve

Ablation test	n_s	av_s	t_{all}	t_{solved}
Big-M formulation	98	38.15	13,555.49	3,700.62
Splitting	98	42.58	11,409.96	4,129.83
Ideal formulation	100	36.02	6,777.95	3,460.17
Splitting+Ideal	100	36.75	8,277.85	3,565.02
Inter Deps	99	30.06	8,561.21	2,916.23
Intra Deps	98	33.03	10,426.82	3,203.54
Inter+Intra Deps	98	25.81	9,729.07	2,503.41
All methods enabled	100	26.52	5,953.46	2,572.90

Table 2: Ablation experiments for MNIST. The average is calculated for the images that are verified in all cases. Similarly, t_{solved} is calculated for the images verified in all cases.

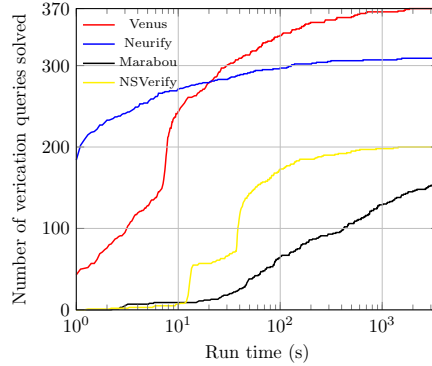


Figure 3: Number of verification queries that Venus, Neurify, Marabou and NSVerify could solve as a function of time.

their effectiveness.

Indeed, the performance gains exhibited by Venus over NSVerify on MNIST and CIFAR-10 are a consequence of combining dependency analysis and ideal formulations. This is evidenced by separately evaluating Venus on MNIST for different combinations of the techniques that the tool implements. Table 2 reports ablation experiments to analyse this in detail. The results confirm that domain splitting is not effective for high dimensional inputs. They also show that ideal formulations and dependency analysis improve on pure big-M formulations not only when they are jointly utilised but also when they are independently employed. In the latter case, ideal formulations enabled the verification of two images that could not be verified by dependency analysis, whereas dependency analysis led to better average and total time for verifying all images that could be verified in all cases. In the latter case the results suggest that the combination of dependency analysis and ideal formulations is preferable in terms of all performance metrics than either technique considered in isolation.

3 Efficient Reachability Analysis for Feed-Forward LE-CPSs with Non-deterministic Environments

This section reports results from the paper “Formal Verification of Neural Agents in Non-deterministic Environments” published at AAMAS 2020. We study the verification problem of agent-environment systems, where the agents are implemented via feed-forward ReLU neural networks and the environment is non-deterministic, against CTL properties. We first show that verifying these systems against (unbounded) reachability properties is undecidable. We then consider 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.

3.1 Neural Agent in Non-deterministic Environment Systems

In this section, for the sake of self-containedness, we introduce formally 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]. Differently from [3], following the dynamism and unpredictability of the environments where autonomous agents are typically deployed [24], 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 a formal description of NANES components: a neural agent and a non-deterministic environment. 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 [28], and is implemented via a ReLU-FFNN or via a piecewise-linear (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 $N_1, \dots, N_h$ computing functions $f_{N_1}, \dots, f_{N_h}$, $h \geq 1$, $prot$ is a PWL combination of the latter.*

When $h = 1$, $prot(s)$ can be defined, e.g., as $f_{N_1}(s)$ for $s \in S$.

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 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 $\mathcal{S} = (Ag, E, I)$ where $Ag = (Act, prot)$ is a neural agent, $E = (S, t_E)$ is an environment, and $I \subseteq S$ is a closed set 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. Note that this does not prevent NANES from modelling a wide class of non-linear environments as these can be approximated to arbitrary precision [8]. 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 $a \in Act$.

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 , $I \subseteq S$, and $T \subseteq R \times R$ is the successor relation defined by $(s, s') \in T$ iff $s' \in t_E(s, prot(s))$.*

In the rest of the section, 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 $\text{paths}(s)$ 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 [7]. Inspired by Real-Time Computation Tree Logic (RTCTL) [10], 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 $bCTL_{\mathbb{R}^{\leq}}$, is defined by the following BNF:*

$$\begin{aligned} \varphi &::= \alpha \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid EX^k \varphi \mid AX^k \varphi, \\ \alpha &::= c_1(d_1) + \dots + c_l(d_l) \geq c, \end{aligned}$$

where $\geq \in \{\leq, \geq\}$, $d_i \in \{1, \dots, m\}$, $c_i, c \in \mathbb{R}$, and $k \in \mathbb{N}$.

Here atomic propositions α are linear constraints on the components of a state. For instance, the atomic proposition $(d_1) + (d_2) \leq 2$ states that “the sum of the d_1 -st and d_2 -nd components is less than or equal to 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”. 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 $E(\varphi U^1 \psi) \triangleq \psi \vee (\varphi \wedge EX^1 \psi)$, and $E(\varphi U^k \psi) \triangleq \psi \vee (\varphi \wedge EX^1 E(\varphi U^{k-1} \psi))$ for $k > 1$, and analogously with $A(\varphi U^k \psi)$.

Although $bCTL_{\mathbb{R}^{\leq}}$ 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.

We now define the logic CTL built from the atoms of $bCTL_{\mathbb{R}^{\leq}}$.

Definition 8 (CTL). *The branching-time logic CTL is defined by the following BNF:*

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

where α is an atomic proposition in $bCTL_{\mathbb{R}^{\leq}}$.

Comparing $\text{bCTL}_{\mathbb{R}^{\leq}}$ to CTL, we observe that on the one hand $AX^k\varphi$ and $EX^k\varphi$ are expressible, respectively, as $AX(\dots(AX\varphi)\dots)$ and $\neg AX(\dots(AX\neg\varphi)\dots)$, where AX is applied k times. On the other hand, CTL includes the AF (“in all paths eventually”) and EU (unbounded until) modalities capable of expressing arbitrary reachability, whereas $\text{bCTL}_{\mathbb{R}^{<}}$ admits bounded specifications only. Note that, while $\text{bCTL}_{\mathbb{R}^{\leq}}$ is clearly less expressive than CTL, it still captures properties of interest.

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

Definition 9 (Satisfaction). *For a model $\mathcal{M}_S$, an environment state s , and a $\text{bCTL}_{\mathbb{R}^{<}}$ formula φ , the satisfaction of φ at s in $\mathcal{M}_S$, denoted $(\mathcal{M}_S, s) \models \varphi$, or simply $s \models \varphi$ when $\mathcal{M}_S$ is clear from the context, is inductively defined as follows:*

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

We assume the usual definition of satisfaction for CTL; this can be given as standard by using the atomic case from Definition 20.

We are interested in the reachability problem, that is, whether there is an initial state from which a specification holds. This is the basis of the verification problem that we define below.

Definition 10 (Reachability problem). *Given a NANES S and a formula φ , determine whether there exists a state $s \in I$ such that $s \models \varphi$.*

Note that we can check *bounded safety* by checking the complement of the reachability problem for the specification $EX^1 \text{unsafe} \vee \dots \vee EX^k \text{unsafe}$, aiming to detect whether an unsafe state is reachable.

3.2 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 2. *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

- $Q = \{1, \dots, k\}$ 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
- $\delta : (Q \setminus \{q_a\}) \times \Sigma \mapsto Q \times \Sigma \times \{L, R\}$ 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 $m \in \{L, R\}$ 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 $m = L$ and right if $m = R$, 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 $\mathcal{S} = ((Act, prot), (S, t_E), I)$ and an unbounded temporal formula φ such that $\mathcal{S} \models \varphi$ iff M halts on $\omega_0 = a_1 \dots a_h$.

- each state of $\mathcal{S}$ 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 i th digit after the dot stores the content of the i th 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 i th digit after the dot stores the content of the i th 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 $m = R$, 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 $m = L$, that is, we pop the top symbol from the left stack and push σ_n onto the right stack.
- φ is the (unbounded) 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 $\mathcal{S}$ and φ are as required.

Conversely, we can construct a NANES $\mathcal{S}' = ((Act', prot'), (S', t'_E), I')$ where t'_E is a linear function and $prot'$ is a PWL function such that $\mathcal{S}' \models \varphi$ iff M halts on ω_0 . Namely,

- $Act' = S' = S$, $I' = I$,

Distribution (A) Approved for public release; distribution is unlimited.

- $t'_E(s, a) = a$, 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 [27].

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

Corollary 1. *Verifying NANES against CTL formulae is undecidable.*

3.3 Bounded CTL

We now proceed to investigate the verification problem for the bounded CTL specification language. We recall 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 (see Lemma 3.1 in WP2.1 Month 18 Deliverable).

Lemma 4. *Given a NANES $\mathcal{S} = ((Act, prot), (S, t_E), I)$ and specification $\varphi \in bCTL_{\mathbb{R}^{\leq}}$, there is a NANES $\mathcal{S}' = ((Act, prot'), (S', t'_E), I')$, 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 $\varphi' \in bCTL_{\mathbb{R}^{<}}$ such that $\mathcal{S} \models \varphi$ iff $\mathcal{S}' \models \varphi'$.*

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 $t_i : \mathbb{R}^{m+n} \rightarrow \mathbb{R}^m$.

Given a MILP program π , we use $vars(\pi)$ to denote the set of variables in π . Denote by $\mathbf{a}$ the *assignment function* $\mathbf{a} : vars(\pi) \rightarrow \mathbb{R}$, which defines the specific (binary, integer or real) value assigned to a MILP program variable. We write $\mathbf{a} \models \pi$ if $\mathbf{a}$ *satisfies* π , i.e., if $\mathbf{a}(\delta) \in \{0, 1\}$ for each binary variable δ , $\mathbf{a}(\iota) \in \mathbb{N}$ for each integer variable ι , and all constraints in π are satisfied. Hereafter, we will 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*.

3.3.1 Monolithic Encoding

We now recall the recursive encoding of the verification problem into a single MILP and the associated monolithic verification procedure, also reported in WP2.1 Month 18 Deliverable. As a stepping stone, we first 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 method, which we denote by $C_i(\mathbf{x}, \mathbf{y})$ (note that $\mathbf{x} \cup \mathbf{y} \subseteq vars(C_i(\mathbf{x}, \mathbf{y}))$). Solutions of $C_i(\mathbf{x}, \mathbf{y})$ represent pairs of consecutive environment states:

Lemma 5. *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 $\mathcal{M}_S$, we have that $s' = t_i(s, prot(s))$ iff there is an assignment $\mathbf{a}$ to $vars(C_i(\mathbf{x}, \mathbf{y}))$ such that $s = \mathbf{a}(\mathbf{x})$, $s' = \mathbf{a}(\mathbf{y})$, and $\mathbf{a} \models C_i(\mathbf{x}, \mathbf{y})$.*

As a second step, given a NANES $\mathcal{S}$ and a formula $\varphi \in bCTL_{\mathbb{R}^{\leq}}$, we construct a MILP program $\pi_{\mathcal{S}\varphi}$, whose feasibility corresponds to the existence of a state in $\mathcal{M}_S$ that satisfies φ . For ease of presentation, and without loss of generality, we assume that φ may contain only the

$$\begin{aligned}
\pi_{\mathcal{S}\alpha}(\mathbf{x}) &= \{C_\alpha(\mathbf{x})\}, \text{ where } C_\alpha(\mathbf{x}) \text{ is defined as } c_1x_{d_1} + \dots + c_lx_{d_l} \geq c \\
&\quad \text{for } \alpha = c_1(d_1) + \dots + c_l(d_l) \geq c \text{ and } \mathbf{x} = (x_1, \dots, x_m), \\
\pi_{\mathcal{S}\varphi_1 \vee \varphi_2}(\mathbf{x}) &= (\delta = 1) \Rightarrow \pi_{\mathcal{S}\varphi_1}(\mathbf{x}) \cup (\delta = 0) \Rightarrow \pi_{\mathcal{S}\varphi_2}(\mathbf{x}), \\
\pi_{\mathcal{S}\varphi_1 \wedge \varphi_2}(\mathbf{x}) &= \pi_{\mathcal{S}\varphi_1}(\mathbf{x}) \cup \pi_{\mathcal{S}\varphi_2}(\mathbf{x}), \\
\pi_{\mathcal{S}EX\varphi}(\mathbf{x}) &= \bigcup_{i=1}^b (\delta_i = 1) \Rightarrow C_i(\mathbf{x}, \mathbf{y}) \cup \{\delta_1 + \dots + \delta_b = 1\} \cup \pi_{\mathcal{S}\varphi}(\mathbf{y}), \\
\pi_{\mathcal{S}AX\varphi}(\mathbf{x}) &= C_1(\mathbf{x}, \mathbf{y}_1) \cup \dots \cup C_b(\mathbf{x}, \mathbf{y}_b) \cup \pi_{\mathcal{S}\varphi}(\mathbf{y}_1) \cup \dots \cup \pi_{\mathcal{S}\varphi}(\mathbf{y}_b),
\end{aligned}$$

Figure 4: Monolithic encoding $\pi_{\mathcal{S}\varphi}$ for $\varphi \in \text{bCTL}_{\mathbb{R}^{\leq}}$, where the binary variables $\delta, \delta_1, \dots, \delta_b$, the state variables $\mathbf{y}_1, \dots, \mathbf{y}_b, \mathbf{y}$, and all auxiliary variables in $C_i(\mathbf{x}, \mathbf{y}_i)$ are fresh.

temporal modalities EX^1 and AX^1 , for which we write EX and AX , respectively¹. We make use of the *indicator constraints* of the form $(\delta = v) \Rightarrow c$, for a binary variable δ , binary value $v \in \{0, 1\}$ and a linear constraint c , with the semantics that whenever the value of δ is v , the constraint c should hold. Indicator constraints are supported by all major commercial MILP solvers and can be seen a 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. For instance, the disjunction $x = 3 \vee y = 5$ can be encoded using two auxiliary binary variables $\delta_i, i \in \{1, 2\}$, and the following set of MILP constraints:

$$(\delta_1 = 1) \Rightarrow x = 3, \quad (\delta_2 = 1) \Rightarrow y = 5, \quad \delta_1 + \delta_2 = 1.$$

Here, the constraint $\delta_1 + \delta_2 = 1$ ensures that at least one of the two clauses is satisfied (note that, in general, the above encoding does not rule out an assignment where both clauses are satisfied at the same time). Given a binary variable δ and a set of constraints π , we abbreviate $\{(\delta = v) \Rightarrow c \mid c \in \pi\}$ to $(\delta = v) \Rightarrow \pi$.

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

Definition 11. *Given a NANES $\mathcal{S}$ and a formula $\varphi \in \text{bCTL}_{\mathbb{R}^{\leq}}$, their monolithic MILP encoding $\pi_{\mathcal{S}\varphi}$ is defined as the MILP program $\pi_{\mathcal{S}\varphi}(\mathbf{x})$, where $\mathbf{x}$ is a tuple of fresh state variables, and $\pi_{\mathcal{S}\varphi}(\mathbf{x})$ is built inductively using the rules in Figure 4.*

Intuitively, in the program $\pi_{\mathcal{S}\varphi}(\mathbf{x})$ the variables $\mathbf{x}$ refer to the states that satisfy the formula φ . In the encoding in Figure 4, the base case $\pi_{\mathcal{S}\alpha}(\mathbf{x})$ 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 $\mathbf{a}$, 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.

¹As discussed in Section 3.1, for $Q \in \{A, E\}$, the formula $QX^k\varphi$ is equivalent to $QX^1(\dots(QX^1\varphi)\dots)$ where QX^1 is applied k times to φ and which grows linearly in k assuming unary encoding of numbers.

- We encode EX via 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 $\pi_{\mathcal{S}\varphi}$ 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 $\pi_{\mathcal{S}\varphi}$, each depending on one of the successor state variables $\mathbf{y}_i$.

Note that the size of $\pi_{\mathcal{S}\varphi}$ may grow exponentially due to b repetitions of $\pi_{\mathcal{S}\psi}$ in $\pi_{\mathcal{S}AX\psi}(\mathbf{x})$; for $\varphi = AX^k\alpha$, the size of $\pi_{\mathcal{S}\varphi}$ is $O(k \cdot b^k \cdot |\mathcal{S}|)$. 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 $\pi_{\mathcal{S}\varphi}$ remains polynomial $O(k \cdot b \cdot |\mathcal{S}|)$.

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

Lemma 6. *Given a NANES $\mathcal{S}$, a formula $\varphi \in bCTL_{\mathbb{R}^{\leq}}$ and a state s in $\mathcal{M}_{\mathcal{S}}$, the following are equivalent:*

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

Proof. Let s be a state in $\mathcal{M}_{\mathcal{S}}$. 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:

$\varphi = \varphi_1 \vee \varphi_2$. Then

$$\begin{aligned}
s \models \varphi_1 \vee \varphi_2 &\Leftrightarrow \\
s \models \varphi_1 \text{ or } s \models \varphi_2 &\Leftrightarrow \\
\text{there exists } \mathbf{a}_1 \text{ s.t. } \mathbf{a}_1 \models \pi_{\mathcal{S}\varphi_1}(\mathbf{x}), \mathbf{a}_1(\delta) = 1 \text{ and } s = \mathbf{a}_1(\mathbf{x}), &\text{ or there exists } \mathbf{a}_2 \text{ s.t. } \mathbf{a}_2 \models \pi_{\mathcal{S}\varphi_2}(\mathbf{x}), \\
\mathbf{a}_1(\delta) = 0 \text{ and } s = \mathbf{a}_2(\mathbf{x}) &\Leftrightarrow \\
\text{there exists } \mathbf{a} \text{ s.t. } \mathbf{a} \models \pi_{\mathcal{S}\varphi_1 \vee \varphi_2}(\mathbf{x}), \text{ and } s = \mathbf{a}(\mathbf{x}). &
\end{aligned}$$

$\varphi = \varphi_1 \wedge \varphi_2$. Then

$$\begin{aligned}
s \models \varphi_1 \wedge \varphi_2 &\Leftrightarrow \\
s \models \varphi_1 \text{ and } s \models \varphi_2 &\Leftrightarrow \\
\text{there exists } \mathbf{a}_1 \text{ s.t. } \mathbf{a}_1 \models \pi_{\mathcal{S}\varphi_1}(\mathbf{x}) \text{ and } s = \mathbf{a}_1(\mathbf{x}), \text{ and there exists } \mathbf{a}_2 \text{ s.t. } \mathbf{a}_2 \models \pi_{\mathcal{S}\varphi_2}(\mathbf{x}) &\text{ and } \\
s = \mathbf{a}_2(\mathbf{x}) &\Leftrightarrow \text{vars}(\pi_{\mathcal{S}\varphi_1}(\mathbf{x})) \cap \text{vars}(\pi_{\mathcal{S}\varphi_2}(\mathbf{x})) = \mathbf{x} \\
\mathbf{a}_1 \cup \mathbf{a}_2 \models \pi_{\mathcal{S}\varphi_1}(\mathbf{x}) \cup \pi_{\mathcal{S}\varphi_2}(\mathbf{x}). &
\end{aligned}$$

$\varphi = EX\varphi_1$. Then

$$\begin{aligned}
s \models EX\varphi_1 &\Leftrightarrow \\
\text{there exists a successor } s' \text{ of } s \text{ in } \mathcal{M}_{\mathcal{S}} \text{ s.t. } s' \models \varphi_1 &\Leftrightarrow \\
\text{there exists } i \in \{1, \dots, k\} \text{ and an assignment } \mathbf{a}_i \text{ such that } \mathbf{a}_i(\delta_i) = 1, \mathbf{a}_i(\delta_j) = 0 \text{ for } j \neq i, & \\
\mathbf{a}_i \models C_i(\mathbf{x}, \mathbf{y}), \mathbf{a}_i(\mathbf{x}) = s \text{ and } \mathbf{a}_i(\mathbf{y}) = s', \text{ and there exists an assignment } \mathbf{a}' \text{ s.t. } \mathbf{a}' \models \pi_{\mathcal{S}\varphi_1}(\mathbf{y}) &\text{ and } \\
\mathbf{a}'(\mathbf{y}) = s' &\Leftrightarrow \\
\text{there exists } \mathbf{a} \text{ s.t. } \mathbf{a} \models \pi_{\mathcal{S}EX\varphi_1}(\mathbf{x}) \text{ and } \mathbf{a}(\mathbf{x}) = s. &
\end{aligned}$$

$\varphi = AX\varphi_1$. Then

$$\begin{aligned}
s \models AX\varphi_1 &\Leftrightarrow \\
\text{for every successor } s_i \text{ of } s \text{ in } \mathcal{M}_{\mathcal{S}} \text{ it holds that } s_i \models \varphi_1, i = 1, \dots, k, &\Leftrightarrow \\
\text{for every successor } s_i \text{ of } s \text{ in } \mathcal{M}_{\mathcal{S}} \text{ it holds that there exists } \mathbf{a}_i \text{ s.t. } \mathbf{a}_i \models \pi_{\mathcal{S}\varphi_1}(\mathbf{y}_i) \text{ and } \mathbf{a}_i(\mathbf{y}_i) = s_i, & \\
i = 1, \dots, k, &\Leftrightarrow \\
\text{there exists } \mathbf{a} \text{ s.t. } \mathbf{a} \models \pi_{\mathcal{S}AX\varphi_1}(\mathbf{x}) \text{ and } \mathbf{a}(\mathbf{x}) = s. &
\end{aligned}$$

□

Algorithm 2 The monolithic MILP verification procedure.

```

1: procedure MONO-VERIFY( $\mathcal{S}, \varphi$ )
2:   Input: NANES  $\mathcal{S} = (Ag, E, I)$ ; formula  $\varphi \in \text{bCTL}_{\mathbb{R}^<}$ 
3:   Output: True/False
4:    $\varphi_I \leftarrow$  Boolean  $\text{bCTL}_{\mathbb{R}^{\leq}}$  formula representing  $I$ 
5:    $\varphi' \leftarrow \varphi \wedge \varphi_I$ 
6:    $\pi_{\mathcal{S}, \varphi'} \leftarrow$  MILP associated with  $\mathcal{S}$  and  $\varphi'$ 
7:    $feasible \leftarrow \text{MILP\_SOLVER}(\pi_{\mathcal{S}, \varphi'})$ 
8:   return  $feasible$ 

```

Finally, we are ready to devise a procedure that solves the verification problem by checking feasibility of the monolithic MILP encoding for the property to be verified together with a restriction to the initial states of $\mathcal{S}$, $\pi_{\mathcal{S} \neg \varphi \wedge \varphi_I}$. A feasible solution to $\pi_{\mathcal{S} \varphi \wedge \varphi_I}$ then provides a witness to the reachability problem. Conversely, infeasibility implies that no witness could be found, and so the property is not satisfied.

The procedure is given by Algorithm 4. 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 $[l_1, u_1] \times \dots \times [l_m, u_m]$ is represented by the formula

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

Therefore, $\varphi' = \varphi \wedge \varphi_I$ is a formula from $\text{bCTL}_{\mathbb{R}^{\leq}}$, so $\pi_{\mathcal{S}, \varphi'}$ is well-defined and can be processed by an MILP solver.

Soundness and completeness of the verification procedure relies on Lemma 6 and is shown by the following.

Theorem 3. *Given a NANES $\mathcal{S}$ and a formula $\varphi \in \text{bCTL}_{\mathbb{R}^<}$, Algorithm 4 returns True iff there exists $s \in I$ such that $s \models \varphi$.*

Proof. Suppose that Algorithm 4 returns *True*. It follows that $\pi_{\mathcal{S} \varphi \wedge \varphi_I}(\mathbf{x})$ is feasible, and there exists an assignment $\mathbf{a}$ to $\text{vars}(\pi_{\mathcal{S} \varphi \wedge \varphi_I}(\mathbf{x}))$ such that $\mathbf{a} \models \pi_{\mathcal{S} \varphi \wedge \varphi_I}(\mathbf{x})$. Take $s = \mathbf{a}(\mathbf{x})$. 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 $\mathcal{M}_{\mathcal{S}}$, and by Lemma 6 we have that $s \models \varphi_I$ and $s \models \varphi$. Conversely, if there exists $s \in I$ such that $s \models \varphi$, we obtain that there is an assignment satisfying $\pi_{\mathcal{S} \varphi \wedge \varphi_I}(\mathbf{x})$, and therefore Algorithm 4 returns *True*. $\square$

Note that by implementing disjunctive cases ($\varphi_1 \vee \varphi_2$ and $EX\varphi$) using auxiliary binary variables, the monolithic encoding creates one large MILP program whose feasibility is then checked to provide an answer to the verification problem. Since this program accounts for all possible evolution branches that need to be considered to verify satisfiability of the given specification, checking its feasibility can be expensive.

3.4 Compositional Encoding and Verification Procedure

We now propose a different encoding that instead of delegating disjunction to the MILP solver creates a separate program for each disjunct. More specifically, for a bounded CTL formula φ , we define a set $\Pi_{\mathcal{S}, \varphi}$ of MILP programs with the property that there exists a state s in $\mathcal{M}_{\mathcal{S}}$ such that $s \models \varphi$ iff at least one of the programs in $\Pi_{\mathcal{S}, \varphi}$ is feasible. A specific feature of this encoding

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

Figure 5: Compositional encoding $\Pi_{\mathcal{S},\varphi}$ for $\varphi \in \text{bCTL}_{\mathbb{R}^{\leq}}$.

lies in its parallelisability. Due to this, it is particularly amenable to efficiently finding bugs that can be reached within a few steps along some of the paths from the initial states, similarly to BMC [6]. We demonstrate this experimentally in the next section after having introduced the encoding here.

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

Definition 12. *Given a NANES $\mathcal{S}$ and a bounded CTL formula φ , their compositional MILP encoding $\Pi_{\mathcal{S},\varphi}$ is defined as the set of MILP programs $\Pi_{\mathcal{S},\varphi}(\mathbf{x})$, where $\mathbf{x}$ is a tuple of fresh state variables, and $\Pi_{\mathcal{S},\varphi}(\mathbf{x})$ is built inductively using the rules in Figure 5.*

Intuitively, in $\Pi_{\mathcal{S},\varphi}(\mathbf{x})$ the variables $\mathbf{x}$ refer to the states that satisfy the formula φ . In Figure 5 $C_\alpha(\mathbf{x})$ is the linear constraint corresponding to the atomic proposition α defined over $\mathbf{x}$. 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. In $\Pi_{\mathcal{S},\varphi}$ 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 $i = 1, \dots, b$.

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

We can prove that $\Pi_{\mathcal{S},\varphi}$ is as intended.

Lemma 7. *Given a NANES $\mathcal{S}$, a bounded CTL formula φ and a state s in $\mathcal{M}_{\mathcal{S}}$, the following are equivalent:*

Distribution (A) Approved for public release; distribution is unlimited.

Algorithm 3 The compositional MILP verification procedure.

```

1: procedure COMP-VERIFY( $\mathcal{S}, \varphi$ )
2:   Input: NANES  $\mathcal{S} = (Ag, E, I)$ ; bounded CTL formula  $\varphi$ 
3:   Output: True/False
4:    $feasible \leftarrow \text{False}$ 
5:    $\varphi_I \leftarrow$  Boolean formula representing  $I$ 
6:    $\varphi' \leftarrow \varphi \wedge \varphi_I$ 
7:    $\Pi_{\mathcal{S}, \varphi'} \leftarrow$  Set of MILPs associated with  $\mathcal{S}$  and  $\varphi'$ 
8:   for  $\pi$  in  $\Pi_{\mathcal{S}, \varphi'}$  do
9:      $aux \leftarrow \text{MILP\_SOLVER}(\pi)$ 
10:    if  $aux$  is True then
11:       $feasible \leftarrow \text{True}$ 
12:      break
13:   return  $feasible$ 

```

1. $s \models \varphi$.

2. There is a MILP $\pi(\mathbf{x}) \in \Pi_{\mathcal{S}, \varphi}(\mathbf{x})$ and an assignment $\mathbf{a}$ to $\text{vars}(\pi(\mathbf{x}))$ such that $s = \mathbf{a}(\mathbf{x})$ and $\mathbf{a} \models \pi(\mathbf{x})$.

Based on Lemma 7, we can devise a verification procedure that searches for a feasible MILP in the set of MILPs generated by the encoding of Figure 5. This procedure is presented in Algorithm 3. Importantly, it naturally lends itself to parallelisation when checking feasibility of the generated programs. In turn this enables us to check in parallel for a possible falsification of formulas in which the temporal operator is universally quantified as in $AX^k\alpha$. As we will see in the next section, this will become particularly useful when verifying bounded safety.

Based on Lemma 7, we can devise a verification procedure that searches for a feasible MILP in the set of MILPs generated by the encoding of Figure 5. This procedure is presented in Algorithm 3. Importantly, it naturally lends itself to parallelisation when checking feasibility of the generated programs. In turn this enables us to check in parallel for possible witnesses to formulas in which the temporal operator is existentially quantified as in $EX^k\alpha$. As we will see in the next section, this will become particularly useful when verifying bounded safety.

3.5 Implementation and Experiments

We have implemented the monolithic and the compositional verification procedures in an open source toolkit called VENMAS [29]. The tool takes as input a bounded CTL specification φ and a NANES $\mathcal{S}$. The top-level call to the tool returns **True** if φ is satisfied by $\mathcal{S}$, and returns **False** if φ fails for some initial state of $\mathcal{S}$. 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 can specify a parameter to determine whether the monolithic or compositional procedure with parallel or sequential execution is to be used. For the monolithic procedure, VENMAS follows the respective procedure from Algorithm 4. For the compositional procedure, VENMAS performs the computation in line 7 of Algorithm 3 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 (on the same machine) retrieving tasks from the jobs queue: one worker process under sequential execution and multiple (8, by default) worker processes under parallel execution. The *main* process finishes either when a MILP-solving job terminates with a feasible solution (i.e., a counter-example), or all the jobs gave infeasible results, or no result was returned within a given time limit.

In order to produce more optimal MILP formulations, both in the case of monolithic and compositional encodings, we compute valid lower and upper bounds for all state and action variables starting from the bounds of the input states given by I . Through the networks the bounds are propagated using symbolic linear relaxation² [31] and are used in the Big-M encoding of the ReLU nodes. For other kind of constraints (e.g., the transition functions in the usecases below), 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 Figure 5 requires computing the whole set $\Pi_{S,\varphi}$ 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 have to idle in the beginning. Therefore, for the experiments, we have implemented and used a version of the compositional encoding that accepts the fragment of bounded CTL consisting of (arbitrary) disjunction, conjunction over atomic propositions and existential path quantifier EX , and that computes the programs in $\Pi_{S,\varphi}$ and adds them to the jobs queue one by one in the depth-first fashion. The worker processes then can start verification 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 early, even before all subproblems have been computed. Additionally, we integrated a further optimisation, where we can discard an individual job (hence, avoid adding it to the jobs queue) by looking at 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 job needs to be created. Finally, we note that disjunction of atomic propositions is 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.

We used Python to implement the tool and relied on Gurobi ver. 9.0 [13] as a back-end to resolve the feasibility of the generated MILP problems. To evaluate our tool, we performed experiments on a machine with an Intel Core i7-7700K CPU @ 4.20GHz with 16GB of RAM, running a 64-bit version of Ubuntu 20.04. We now describe the two usecases we evaluated our tool on and present the obtained experimental results.

3.6 FrozenLake Example

We started by validating our toolkit on a variant of the FROZENLAKE scenario [25], where an agent navigates in a grid world consisting of walkable tiles (frozen surface) and non-walkable ones (hole in the ice) leading to the agent falling into water. The purpose of the agent is to arrive at the goal tile while avoiding the holes. At each step the agent chooses a direction to walk, but since walkable tiles are slippery, the movement direction is uncertain and only partially depends on the chosen direction. Namely, the agent can end up moving in any of the three directions: the chosen one, to the left or to the right of it, see Figure 6 for an illustration.

3.6.1 NANES Encoding and Specifications

We now formalise this scenario as follows.

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

²Symbolic linear relaxation 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 allow to capture implicit relationship between nodes of intermediate layers, resulting in reasonably tight bounds that can be computed quickly.

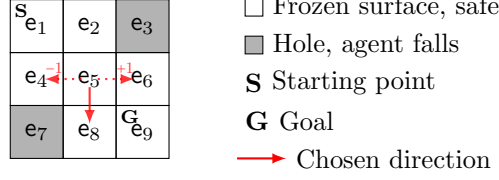


Figure 6: The FROZENLAKE surface.

- $Act_{\Pi} = \{d_1, d_2, d_3, d_4\} \subseteq \mathbb{R}^4$ encodes the directions left, down, right and up, where d_i is the vector with 1 at i -th position and 0 everywhere else, and
- $prot_{\Pi}$ 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 $E_{\Pi} = (S_{\Pi}, t_{E_{\Pi}})$, where:

- $S_{\Pi} = \{e_1, e_2, \dots, e_9\} \subseteq \mathbb{R}^9$, where e_i is the vector with 1 at i -th position and 0 everywhere else, and
- $t_{E_{\Pi}}(s, a) = \{mv(s, a_{-1}), mv(s, a), mv(s, a_{+1})\}$,

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., $a_{-1} = d_1$ and $a_{+1} = d_3$ for $a = d_2$). 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 $I_{\Pi} = \{e_1\}$ and the FROZENLAKE system as $\mathcal{S}_{\Pi} = (Ag_{\Pi}, E_{\Pi}, I_{\Pi})$.

We are interested in assessing the safety of the scenario in terms of the following specifications:

$$\varphi_{\text{unsafe}}^k = EX^1 \mathbf{hr} \vee \dots \vee EX^k \mathbf{hr}$$

and

$$\varphi_{\text{fail}}^k = EX^1 \mathbf{hr} \vee \dots \vee EX^{k-1} \mathbf{hr} \vee EX^k \mathbf{goalmissed},$$

for various values of k , where $\mathbf{hr} \triangleq \bigvee_{h \in \{3,7\}} (h) > 0.5$ (i.e., holes voided) and $\mathbf{goalmissed} \triangleq (9) < 0.5$. The formula $\varphi_{\text{unsafe}}^k$ states that there exists an evolution of the system in which the agent reaches a hole within the first k steps. The formula φ_{fail}^k states that there exists an evolution of the system where either the agent reaches a hole within the first $k-1$ steps or misses the goal state at the k -th step. Intuitively, when the former is not satisfied, it means that all k -bounded runs are safe (no hole is reached), while when the latter is not satisfied, it means that all k -bounded runs are safe and successful (the goal is reached in the end state).

3.6.2 Implementation and Experiments

We considered the grid world as in Figure 6, that is, there are two holes, tiles e_3 and e_7 , the initial state is tile e_1 , and the goal is tile e_9 .

We trained the agent's neural network using an actor-critic [28] approach. 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, 0 otherwise. So, the agent is trained to maximise the chance of reaching

Table 3: Verification times for the properties $\varphi_{\text{unsafe}}^k$ and φ_{fail}^k for different values of k on the FROZENLAKE scenario encoded as a NANES. Greyed-out cells indicate a **True** result, otherwise a **False** result. A hyphen ‘-’ represents a one hour timeout.

	MONOLITHIC		COMP-SEQ		COMP-PAR		NAIVE-COMP-PAR	
k	$\varphi_{\text{unsafe}}^k$	φ_{fail}^k	$\varphi_{\text{unsafe}}^k$	φ_{fail}^k	$\varphi_{\text{unsafe}}^k$	φ_{fail}^k	$\varphi_{\text{unsafe}}^k$	φ_{fail}^k
1	0.04	0.04	0.03	0.06	0.04	0.07	0.08	0.07
2	0.07	0.08	0.04	0.09	0.07	0.12	0.33	0.10
3	0.53	0.24	0.14	0.17	0.12	0.24	0.95	0.30
4	1.67	0.58	0.18	0.25	0.19	0.30	3.38	0.40
5	9.50	0.97	0.41	0.26	0.43	0.35	12.75	0.43
6	241.41	2.75	1.16	0.29	1.16	0.35	47.26	0.59
7	915.34	8.61	3.37	0.32	3.41	0.38	168.13	0.80
8	3476.08	8.70	10.33	0.35	10.30	0.42	589.66	1.64
9	–	5.29	32.50	0.36	32.68	0.46	2065.32	4.27
10	–	30.97	111.22	0.38	112.87	0.46	–	13.06

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.

We verified the FROZENLAKE system $\mathcal{S}_{\text{fl}}$ against the specifications $\varphi_{\text{unsafe}}^k$ and φ_{fail}^k . To achieve more optimal encodings, the specifications were equivalently formulated to minimise the number of EX operators:

$$\begin{aligned}
\varphi_{\text{unsafe}}^k &= \underbrace{EX(\text{hr} \vee EX(\text{hr} \vee \dots (\text{hr} \vee EX\text{hr})))}_{k \text{ } EX\text{'s}}, \\
\varphi_{\text{fail}}^k &= \underbrace{EX(\text{hr} \vee EX(\text{hr} \vee \dots (\text{hr} \vee EX\text{goalmissed})))}_{k \text{ } EX\text{'s}},
\end{aligned}$$

The experiments showed that it was impossible to reach holes by following agent’s protocol function, and that there were runs where the goal was missed.

In Table 3, we report the amount of time (in seconds) taken to resolve the specifications $\varphi_{\text{unsafe}}^k$ and φ_{fail}^k 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. We also report the results for the compositional procedure implementing the original compositional encoding in parallel execution, denoted NAIVE-COMP-PAR. 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 witnesses to φ_{fail}^k and at proving that $\varphi_{\text{unsafe}}^k$ is not satisfied. There was no difference between the sequential and parallel executions when verifying $\varphi_{\text{unsafe}}^k$ because all jobs have been discarded by the splitting process. As for naive versus optimised implementations of compositional encoding, we see that the naive version is significantly slower for proving safety as all the subproblems are passed to be processed by Gurobi. Moreover when finding witnesses for bigger values of k we start seeing the overhead resulting from the need to compute first the whole set $\Pi_{\mathcal{S}, \varphi_{\text{fail}}^k}$ before dispatching the problems to the workers.

The monolithic procedure also managed to locate witnesses quickly. As for proving safety, it was significantly slower than the compositional procedure for $k \geq 6$. The main reason for that was 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 state variables $\mathbf{y}$ when encoding $EX\varphi$ since we do not know in advance which of the b successors they refer to. It is further exacerbated by the discreet nature of the state space in this scenario. For instance, assume that we are in the state $\mathbf{e}_5$, 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 $\mathbf{e}_8$, $\mathbf{e}_4$ and $\mathbf{e}_6$, and the bounds “explode”: 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. When such loose bounds are propagated through the network and other constraints, they quickly become very conservative resulting in MILP formulations that cannot be efficiently solved.

In the next set of experiments, we will see a different picture where the monolithic procedure performs much better when the bounds do not deteriorate so quickly.

3.7 Aircraft Collision Avoidance System Example

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 referred to as VerticalCAS [15]. The intruder is assumed to follow a constant horizontal trajectory. VerticalCAS once every second issues vertical climb/descent 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 climb/descent 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, with 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 $S_{\text{vcas}} = [-3000, 3000] \times [-2500, 2500] \times [0, 40] \times [9]$. A tuple $(h, h_0, \tau, \text{adv}) \in S_{\text{vcas}}$ 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 climb/descent.
- 3) τ (s) is the time to loss of horizontal separation.
- 4) adv is the previous advisory issued by VerticalCAS.

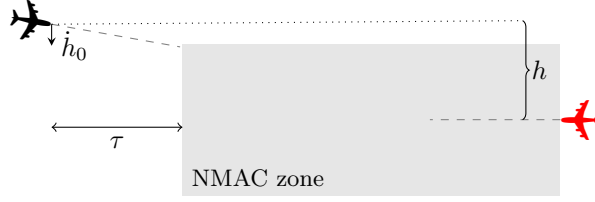


Figure 7: VerticalCAS encounter geometry

Figure 7 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 $F = \{(f_{N_i} : \mathbb{R}^3 \rightarrow \mathbb{R}^9) : i \in [9]\}$, 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.

3.7.1 NANES Encoding and Specification

We now formalise the VERTICALCAS scenario as a NANES $\mathcal{S}_{\text{vcas}} = (Ag_{\text{vcas}}, E_{\text{vcas}}, I_{\text{vcas}})$.

We model VerticalCAS as the neural agent $Ag_{\text{vcas}} = (Act_{\text{vcas}}, prot_{\text{vcas}})$ with the set of actions $Act_{\text{vcas}} = [9]$ and the protocol function producing an action corresponding to the highest-scoring advisory formally defined as $prot_{\text{vcas}}(s) = \arg \max(\text{apply}(\text{select}(s), s))$, where for a state $s = (h, \dot{h}_0, \tau, \text{adv}) \in S_{\text{vcas}}$:

- **select**: $S_{\text{vcas}} \rightarrow F$ selects the neural network corresponding to the previous advisory adv , $\text{select}(s) = f_{\text{adv}}$;
- **apply**: $F \times \mathbb{R}^4 \rightarrow \mathbb{R}^9$ computes the output of a neural network for a given state, $\text{apply}(f, s) = f(h, \dot{h}_0, \tau)$,
- **arg max**: $\mathbb{R}^9 \rightarrow [9]$ 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 $\mathcal{S}_{\text{vcas}}$, defined as $E_{\text{vcas}} = (S_{\text{vcas}}, t_{E_{\text{vcas}}})$. Thus, the environment transition function $t_{E_{\text{vcas}}}$ "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 climb rate $\dot{h}_0$ of the ownship. We bound the number of possible successor states of $t_{E_{\text{vcas}}}$ by 3, that is $b = 3$, so the set of next possible accelerations is $Acc_{\text{adv}'}^{\dot{h}_0} = \{\ddot{h}_0^{(1)}, \ddot{h}_0^{(2)}, \ddot{h}_0^{(3)}\}$ defined as follows. If the current vertical climb rate $\dot{h}_0$ of the ownship is *compliant* with the advisory, the pilot maintains a constant climb rate, 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, $Acc_{\text{DND}}^{\dot{h}_0} = \{\frac{g}{4}, \frac{7g}{24}, \frac{g}{3}\}$.

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

Table 4: Verification times for a VerticalCAS system against the property φ^k for different values of k and $\dot{h}_0$. Greyed-out cells indicate a **True** result, otherwise a **False** result. We use dashes ‘-’ to indicate a one hour timeout.

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	
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	-	

for $t_{E_{\text{vcas}}}$ as:

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

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 taking into account the state transition dynamics.

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

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

for $\dot{H}_0 = \{-19.5 - 3k \mid k = 0, \dots, 8\} \cup \{-39, -39.5, -40\}$ the set of all initial climb rates. 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, it is possible for the ownship to enter the *unsafe region* ($|h| \leq 100$), doing which may potentially lead to an NMAC for small values of τ . So we consider the safety specifications

$$\varphi^k = EX^k ((1) \leq 100 \wedge (1) \geq -100)$$

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.

3.7.2 Implementation and Experiments

Our implementation of VerticalCAS system aims to compute tightest possible bounds for all state and intermediate variables that allow to produce more 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 including the encodings of the networks f_{N_i} 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 0), then only the encoding of f_{N_0} is included. Additionally, this 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 acceleration depending on 9 possible values of the advisory. Again, we compute tight acceleration bounds by looking at the bounds of the new advisory. In particular, when the latter are exact (i.e., we know the exact value of the advisory), the former are exact as well. Tight acceleration bounds allow then for computing tight bounds for the next state, and so on.

We verified the VerticalCAS system $\mathcal{S}_{\text{vcas}}$ 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 $\frac{9}{4} \text{ft/s}^2$ so as to climb to avoid collision with the intruder. The descent rate was not

Table 5: Comparison between naive and optimised implementations of compositional encoding under parallel execution.

	NAIVE-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
φ^1	0.11	0.10	0.10	0.10	0.10	0.09	0.09	0.09	0.09	0.10	0.09	0.09
φ^2	0.39	0.38	0.32	0.31	0.29	0.29	0.31	0.29	0.28	0.26	0.30	0.34
φ^3	0.85	0.34	0.30	0.32	0.32	0.31	0.32	0.31	0.30	0.32	0.34	0.35
φ^4	2.90	2.95	0.30	0.36	0.31	0.38	0.32	0.31	0.38	0.31	0.38	0.40
φ^5	10.54	10.42	10.42	0.35	0.41	0.40	0.41	0.42	0.41	0.34	0.35	0.46
φ^6	37.18	36.88	37.76	37.15	0.47	0.49	0.49	0.41	0.42	0.46	0.48	0.46
φ^7	132.72	130.53	131.01	132.65	132.20	0.50	0.59	0.53	0.65	0.66	0.66	0.83
φ^8	452.56	453.82	564.27	566.90	446.10	453.15	0.84	1.71	1.14	1.86	1.94	2.56
φ^9	1917.49	1599.98	3301.75	3290.76	1542.29	1571.00	2584.51	3435.65	3444.49	-	5.43	6.56
φ^{10}	-	-	-	-	-	-	-	-	-	-	-	-
	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
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	-

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.

In Table 6, we report the performance of the tool in terms of the amount of time (in seconds) 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 witnesses. We attribute it to the fact that, unlike in the FROZENLAKE scenario, 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.

In Table 5, we report the comparison between the naive and optimised implementations of the compositional procedure. We see similar results to the FROZENLAKE experiments.

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 get discarded during the compositional encoding and the workers get to do MILP-solving jobs, so we can notice that parallel execution is approximately twice faster than sequential ex-

ecution. 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 $\dot{h}_0 = -40.5$, $k = 9$).

4 Verifying Strategic Abilities of Neural-symbolic Multi-agent Systems

This section reports results from the paper “Verifying Strategic Abilities of Neural-symbolic Multi-agent Systems” published at KR 2020. We investigate the problem of verifying the strategic properties of multi-agent systems equipped with machine learning-based perception units. We introduce a novel model of agents comprising both a perception system implemented via feed-forward neural networks and an action selection mechanism implemented via traditional control logic. We define the verification problem for these systems against a bounded fragment of alternating-time temporal logic. We translate the verification problem on bounded traces into the feasibility problem of mixed integer linear programs and show the soundness and completeness of the translation. We show that the lower bound of the verification problem is PSPACE and the upper bound is coNEXPTIME. We present a tool implementing the compilation and evaluate the experimental results obtained on a complex scenario of multiple aircraft operating a recently proposed prototype for air-traffic collision avoidance.

4.1 Neural-symbolic Multi Agent Systems

We start by presenting the semantics of neural MAS called neural interpreted systems (NIS). NIS extend *interpreted systems* (IS) [11], a standard semantics for MAS, previously used to reason about the strategic interplay in MAS [23]. Differently from IS, which assume purely symbolic agents, NIS uniquely combine connectionist with symbolic models; agents are endowed with a neural-based perception unit, used to gather information from the environment, and act upon their observations via traditional symbolic control logic. NIS allow for the natural modelling of hybrid architectures combining traditionally programmed control units with perception units synthesised from data. Autonomous vehicles, for instance, use this architecture to neurally interpret the images from their cameras and symbolically decide the required steering angle of the vehicle on the basis of said interpretation [9]. They can be seen as a loosely-coupled neural-symbolic agent model [12].

We begin with a MAS composed of K *neural-symbolic agents* $Agt = \{1, \dots, K\}$ acting in an environment E . We often treat the environment as a special agent thereby considering a MAS as composed of the set $Agt \cup \{E\}$ of agents. Each agent $a \in Agt$ is described by the following.

Definition 13 (Neural-symbolic Agent). *A neural-symbolic agent, or agent, is a tuple $a = (L_a, obs_a, Act_a, prot_a, tr_a)$, where*

- $L_a = Prv_a \times Per_a$ is a nonempty (possibly infinite) set of local states. A local state is a pair (prv_a, per_a) of a private state $prv_a \in Prv_a$ and a percept $per_a \in Per_a$ storing the state of the environment observed by the agent.
- $obs_a: L_a \times L_E \rightarrow Per_a$ is an observation function mapping pairs of local states $L_a \subseteq \mathbb{R}^h \times \mathbb{R}^n$ and environment states $L_E \subseteq \mathbb{R}^m$ to percepts $Per_a \subseteq \mathbb{R}^n$ and is implemented via a PWL FFNN³ $f_a: \mathbb{R}^{h+n+m} \rightarrow \mathbb{R}^n$.

³Note that in practice the observation function may be implemented via a PWL combination of a number of PWL FFNNs.

- Act_a is a nonempty finite set of actions,
- $prot_a: L_a \rightarrow 2^{Act_a} \setminus \{\emptyset\}$ is a local protocol function defining the actions the agent can perform in a local state.
- $tr_a: L_a \times Act_1 \times \dots \times Act_K \times Act_E \rightarrow Prv_a$ is a local transition function determining the next private state given a local state and a tuple of actions performed by all agents.

The environment's description is similar to the agents' descriptions but has no percepts and observation function.

Definition 14 (Environment). *The environment is a tuple $E = (L_E, Act_E, prot_E, tr_E)$, where $L_E \subseteq \mathbb{R}^m$ is a nonempty (possibly infinite) set of local states, Act_E is a finite set of actions, $prot_E: L_E \rightarrow 2^{Act_E} \setminus \{\emptyset\}$ is a protocol function and $tr_E: L_E \times Act_1 \times \dots \times Act_K \times Act_E \rightarrow L_E$ is a local transition function.*

Similarly to IS, the agents in a NIS are only aware of their local state. Previous proposals have used perception modules [32] implemented symbolically. Here instead the agents in a NIS observe the state of the environment via the neural observation function described above. As a FFNN is a real vector-valued function, we encode the environment's set of states and the agents' sets of percepts as subsets of $\mathbb{R}^m$ and $\mathbb{R}^n$. The agents' private states and percepts encapsulate all the information accessible to the agents' decision making process; we call a pair of a private state and a percept a *local state*. In particular, given the current local state $\ell_a = (prv_a, per_a)$ for each $a \in Agt$ and the current local state ℓ_E of the environment, the operational cycle of the agents is described as follows. First, every agent and the environment chooses a protocol-compliant action α_a , i.e., $\alpha_a \in prot_a(\ell_a)$ for $a \in Agt \cup \{E\}$. The agents synchronously perform the selected actions. Then, every agent $a \in Agt$ updates the private part of its local state according to its local transition function to $prv'_a = tr_a(\ell_a, \alpha_1, \dots, \alpha_K, \alpha_E)$ producing intermediate local state $\ell'_a = (prv'_a, per_a)$. Similarly, the environment updates its local state to $\ell'_E = tr_E(\ell_E, \alpha_1, \dots, \alpha_K, \alpha_E)$. Finally, every agent $a \in Agt$ observes the updated state of the environment and with the information stored in its updated local state generates a percept $per'_a = obs_a(\ell'_a, \ell'_E)$ with which it updates the perception part obtaining the new local state $\ell''_a = (prv'_a, per'_a)$.

We hereafter assume that the local transition functions can be expressed as PWL functions. Note that this does not prevent NIS from modelling a wide spectrum of real-world use cases: if the transition function has a degree of nonlinearity, it can be approximated by a PWL FFNN to an arbitrary level of precision [8].

We now proceed to define the NIS comprising both the agents and the environment, and their associated temporal models. For the definitions we will require the notions of *global states* and *joint actions*. A *global state* $q = (\ell_1, \dots, \ell_K, \ell_E)$ is a tuple of local states for all the agents in the system; it describes the system at a particular instant of time. Given $\ell_a = (prv_a, per_a)$ for $a \in Agt$, a global state $q = (\ell_1, \dots, \ell_K, \ell_E)$ and an agent a , we write $lprv_a(q)$, $lper_a(q)$ and $loc_a(q)$ to denote the private part $lprv_a(q) = prv_a$ and the perception part $lper_a(q) = per_a$ of the local state $loc_a(q) = \ell_a$ of agent a in q . For a global state q and a local protocol $prot_a$ we often write $prot_a(q)$ to mean the set of actions that can be performed by agent a at q , i.e., $prot_a(loc_a(q))$. The set $G = L_1 \times \dots \times L_K \times L_E$ of possible global states is the Cartesian product of the agents' sets of local states. A *joint action* $\alpha = (\alpha_1, \dots, \alpha_K, \alpha_E)$ is a tuple of local actions for all the agents in the system. The set $ACT = Act_1 \times \dots \times Act_K \times Act_E$ of all possible joint actions is the Cartesian product of the agents' sets of local actions. The local transition functions induce a *global transition function* $tr: G \times ACT \rightarrow G$ where, for $q \in G$, $(\alpha_1, \dots, \alpha_K, \alpha_E) \in ACT$, $tr(q, \alpha_1, \dots, \alpha_K, \alpha_E) = q'$ iff

- $\alpha_E \in \text{prot}_E(q)$ and $\text{tr}_E(\text{loc}_E(q), \alpha_1, \dots, \alpha_K, \alpha_E) = \text{loc}_E(q')$; i.e., the environment's action is protocol compliant and its local state is updated as per its local transition function.
- $\alpha_a \in \text{prot}_a(q)$, $\text{tr}_a(\text{loc}_a(q), \alpha_1, \dots, \alpha_K, \alpha_E) = \text{lprv}_a(q')$ and $\text{obs}_a((\text{lprv}_a(q'), \text{lper}_a(q)), \text{loc}_E(q')) = \text{lper}_a(q')$ for every agent $a \in \text{Agt}$; i.e., the agent's action is protocol compliant, the private part of its local state is updated as per its local transition function and the perception part of its local state is updated as per its observation function.

Definition 15 (Neural Interpreted System). *Given a set of agents Agt , an environment E and a set of atomic propositions AP , a neural interpreted system (NIS) is a tuple $\mathcal{S} = (\text{Agt} \cup \{E\}, I, V)$, where:*

- $I \subseteq G$ is a linearly definable set of initial global states.
- $V: AP \rightarrow 2^G$ is a labelling function specifying the atomic propositions that are true in each global state.

The labelling function in a NIS is hereafter assumed to be expressible as a PWL function. In particular, for each $p \in AP$, $V(p)$ returns a description of a subset of G that is expressible as a Boolean PWL function.

Example 2. Consider as a running example two pilots operating two aircraft each equipped with an airborne traffic collision avoidance system (CAS) implemented via PWL FFNNs. Each CAS at every second will issue an advisory to the respective ownship and intruder pilots. The pilot is required to respond to the advisory by applying an acceleration to the aircraft in order to have the aircraft's climb rate compliant with the issued advisory. Each pilot can be modelled as a neural-symbolic agent a , where the pilot's acceleration response is modelled as non-deterministic choice defined in a protocol function prot_a , and local states L_a encode information relevant to each pilot – the aircraft's climb rate as its private state and the previous advisory issued by the CAS as its percept. The remaining physical transition dynamics of the system are modelled in an environment agent E .

NIS are (uniquely) associated with temporal models which can be used to interpret our specification language. These are defined as follows.

Definition 16 (Model). *Given a NIS $\mathcal{S} = (\text{Agt} \cup \{E\}, I, V)$, the induced model of $\mathcal{S}$, or simply the model, is a tuple $\mathcal{M}_{\mathcal{S}} = (G, ACT, T, V)$, where:*

- G is the set of global states and ACT is the set of joint actions of $\mathcal{S}$.
- $T \subseteq G \times ACT \times G$ is the transition relation defined as $(q, \alpha, q') \in T$ iff $\text{tr}(q, \alpha) = q'$.
- V is a labelling function as in Definition 15.

A *path* in a model $\mathcal{M}_{\mathcal{S}} = (G, ACT, T, V)$ is an infinite sequence of global states and joint actions $q^0 \alpha^0 q^1 \alpha^1 \dots$ such that for every $i \geq 0$, $(q^i, \alpha^i, q^{i+1}) \in T$.

We verify NIS against a bounded variant of a restricted subset of ATL^* [4], drawing inspiration from Real Time Computation Tree Logic (RTCTL) [10]. The satisfaction status of the formulae expressible in our language depends only on paths of a bounded length. This has been shown to be practically relevant by enabling the efficient identification of shallow bugs in a system's execution [5, 26]. In addition to this practical consideration, our restriction to a bounded fragment of ATL^* follows the undecidability of the verification problem for unbounded formulae [1].

Definition 17 (Bounded ATL*). *Given a set of agents Agt , an environment E and a set of atomic propositions AP , we define the specification language bATL^* with the following BNF:*

$$\varphi ::= p \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \langle\langle \Gamma \rangle\rangle X^k \varphi \mid [\Gamma] X^k \varphi,$$

where $p \in AP$, $k \in \mathbb{N}$ and $\Gamma \subseteq \text{Agt} \cup \{E\}$.

The formula $\langle\langle \Gamma \rangle\rangle X^k \varphi$ is read as “the group of agents Γ can bring about φ at the k -th step”. The formula $[\Gamma] X^k \varphi$ is read as “irrespective of the actions of the agents in group Γ , φ will hold at the k -th time step.” We inductively abbreviate bounded until $\langle\langle \Gamma \rangle\rangle (\varphi U^k \psi)$ as

$$\begin{aligned} \langle\langle \Gamma \rangle\rangle (\varphi U^1 \psi) &\triangleq \psi \vee (\varphi \wedge \langle\langle \Gamma \rangle\rangle X \psi); \\ \langle\langle \Gamma \rangle\rangle (\varphi U^k \psi) &\triangleq \psi \vee (\varphi \wedge \langle\langle \Gamma \rangle\rangle X (\langle\langle \Gamma \rangle\rangle (\varphi U^{k-1} \psi))), \end{aligned}$$

with the meaning of “the group of agents Γ can bring about ψ at some point in the k following steps and φ holds until then”; the formula $[\Gamma] (\varphi U^k \psi)$ can be defined analogously.

Example 3. For our collision avoidance example we can specify a property expressing that the ownship agent has a strategy to ensure that the two aircraft are safe from a collision after k steps. Given two neural-symbolic agents $\text{Agt} = \{\text{own}, \text{int}\}$ referring to the ownship and intruder defined in Example 2 and taking $\Gamma = \{\text{own}\}$, we can define such a property as $\varphi^k = \langle\langle \text{own} \rangle\rangle X^k \text{safe}$, where **safe** is an atomic proposition representing a safe region.

We now proceed to define the satisfaction relation for bATL^* specifications on NIS-induced models. This uses the notion of a *strategy*.

Definition 18. A strategy for an agent $a \in \text{Agt} \cup \{E\}$ is a function $s_a: L_a \rightarrow 2^{\text{Act}_a}$ s.t. if $\alpha \in s_a(\ell_a)$, then $\alpha \in \text{prot}_a(\ell_a)$.

The strategies defined above follow the memoryless, incomplete information and non-uniform setting from [23] as implemented in the toolkit MCMAS. We can combine a collection of strategies to define a *joint strategy* for a coalition of agents.

Definition 19 (Joint Strategy). Let $\Gamma \subseteq \text{Agt} \cup \{E\}$ be a coalition of agents. A joint strategy is a tuple $s_\Gamma = (s_1, s_2, \dots, s_{|\Gamma|})$ of strategies s_a for each agent $a \in \Gamma$.

We can now define the evaluation of bATL^* formulae on NIS-induced models. Given a set of agents Γ and a global state q , we write q_Γ for the tuple of local states in q restricted to the agents in Γ . Similarly, for a joint action α_Γ , we write α_Γ for the tuple of local actions in α restricted to the agents in Γ . We write $(\mathcal{M}_S, q) \models \varphi$ to mean the bATL^* formula φ is satisfied at state q in $\mathcal{M}_S$; this is often abbreviated $q \models \varphi$.

Definition 20 (Satisfaction). For a model $\mathcal{M}_S = (G, \text{ACT}, T, V)$, a global state q^0 , and a bATL^* formula φ , the satisfaction of φ at q^0 , denoted $q^0 \models \varphi$, is defined as follows:

$$\begin{aligned} q^0 \models p &\text{ iff } q^0 \in V(p), \text{ for } p \in AP; \\ q^0 \models \varphi \vee &\text{ iff } q^0 \models \varphi \text{ or } q^0 \models \psi; \\ q^0 \models \varphi \wedge &\text{ iff } q^0 \models \varphi \text{ and } q^0 \models \psi; \\ q^0 \models \langle\langle \Gamma \rangle\rangle X^k \varphi &\text{ iff there is a joint strategy } s_\Gamma \text{ and an action } \alpha_\Gamma^0 \in s_\Gamma(q_\Gamma^0) \text{ such that all states } q^1 \\ &\text{ with } (q^0, \alpha_\Gamma^0, q^1) \in T \text{ are such that there is an action } \alpha_\Gamma^1 \in s_\Gamma(q_\Gamma^1) \text{ such that all states } q^2 \\ &\text{ with } (q^1, \alpha_\Gamma^1, q^2) \in T \text{ are such that, etc., up to state } q^k, \text{ and we have that } q^k \models \varphi; \\ q^0 \models [\Gamma] X^k \varphi &\text{ iff for all joint strategies } s_\Gamma \text{ and actions } \alpha_\Gamma^0 \in s_\Gamma(q_\Gamma^0) \text{ there is a state } q^1 \text{ with} \\ &(q^0, \alpha_\Gamma^0, q^1) \in T \text{ such that for all actions } \alpha_\Gamma^1 \in s_\Gamma(q_\Gamma^1) \text{ there is a state } q^2 \text{ with } (q^1, \alpha_\Gamma^1, q^2) \in T, \\ &\text{etc., up to } k, \text{ and we have that } q^k \models \varphi. \end{aligned}$$

A specification φ is said to be *satisfied* by a NIS $\mathcal{S}$, denoted $\mathcal{S} \models \varphi$, iff $(\mathcal{M}_{\mathcal{S}}, q) \models \varphi$ for every $q \in I$. This forms the basis of the following verification problem:

Definition 21 (Verification Problem). *Given a NIS $\mathcal{S}$ and a specification $\varphi \in \text{bATL}^*$, determine whether $\mathcal{S} \models \varphi$.*

4.2 The Verification Problem

We here develop decision procedures for the verification problem of Definition 21. These extend the procedures devised for a neural (non-symbolic) agent operating on a non-deterministic environment system (NANES) against a bounded fragment of CTL [1]. As in previous work the approach is based on recasting the verification problem as the feasibility problem of a particular mixed-integer linear program; this is possible due to the fact that NIS are based on PWL functions. Differently from previous work, however, we here need to deal with strategy quantifications, and agent architectures that include neural and symbolic components. We start by observing a logical equivalence that applies to bATL^* .

4.2.1 Normal form of specifications.

We show that specifications in bATL^* can be rewritten using $\langle\langle \Gamma \rangle\rangle X$ and $\llbracket \Gamma \rrbracket X$ only.

Lemma 8. *Given a NIS $\mathcal{S}$, its induced model $\mathcal{M}_{\mathcal{S}}$, a state $q \in G$, and a formula $\varphi \in \text{bATL}^*$, we have that:*

$$q \models \langle\langle \Gamma \rangle\rangle X^k \varphi \quad \text{iff} \quad q \models \underbrace{\langle\langle \Gamma \rangle\rangle X \cdots \langle\langle \Gamma \rangle\rangle X}_{k \text{ repetitions}} \varphi \quad (1)$$

$$q \models \llbracket \Gamma \rrbracket X^k \varphi \quad \text{iff} \quad q \models \underbrace{\llbracket \Gamma \rrbracket X \cdots \llbracket \Gamma \rrbracket X}_{k \text{ repetitions}} \varphi \quad (2)$$

Proof. We prove clause (1), clause (2) can be proved similarly. Let $q^0 = q$. We have that $q^0 \models \langle\langle \Gamma \rangle\rangle X^k \varphi$ iff there is a joint strategy s_{Γ} and an action $\alpha_{\Gamma}^0 \in s_{\Gamma}(q_{\Gamma}^0)$ such that all states q^1 with $(q^0, \alpha^0, q^1) \in T$ are such that there is an action $\alpha_{\Gamma}^1 \in s_{\Gamma}(q_{\Gamma}^1)$ such that all states q^2 with $(q^1, \alpha^1, q^2) \in T$ are such that, etc., up to state q^k , and we have that $q^k \models \varphi$. This is iff there is a joint strategy s_{Γ}^0 with $s_{\Gamma}^0(q^0) = \{\alpha^0\}$ such that all states q^1 with $(q^0, \alpha^0, q^1) \in T$ are such that there is a joint strategy $s_{\Gamma}^1(q^1)$ with $s_{\Gamma}^1(q^1) = \{\alpha^1\}$ such that all states q^2 with $(q^1, \alpha^1, q^2) \in T$ are such that, etc., up to state q^k , and we have that $q^k \models \varphi$. The latter holds iff $q^0 \models \langle\langle \Gamma \rangle\rangle X \cdots \langle\langle \Gamma \rangle\rangle X \varphi$. $\square$

Due to Lemma 8, in the following we assume that any specification is written by using $\langle\langle \Gamma \rangle\rangle X$ and $\llbracket \Gamma \rrbracket X$ only.

4.2.2 Notation.

A *variable assignment* $\mathbf{a}$ for a MILP π is a function $\mathbf{a}: \text{vars}(\pi) \rightarrow \mathbb{R}$ that assigns a value to each variable in π . An assignment $\mathbf{a}$ *satisfies* π , denoted $\mathbf{a} \models \pi$, if $\mathbf{a}(\delta) \in \{0, 1\}$ for each binary variable δ , $\mathbf{a}(\iota) \in \mathbb{Z}$ for each integer variable ι , and all constraints in π are satisfied.

When encoding disjunctive cases, it is convenient to use so-called *indicator* constraints. These are constraints of the form $(\delta = v) \rightarrow c$, where δ is a binary variable and $v \in \{0, 1\}$, stating that whenever the value of δ is v , the linear constraint c should hold. For instance, the two indicator constraints $(\delta_1 = 1) \rightarrow x = 3$ and $(\delta_2 = 1) \rightarrow x = 5$ with the constraint $\delta_1 + \delta_2 = 1$ impose that the value of x is either 3 or 5. Given a binary variable δ and a set of MILP constraints π , we write $(\delta = v) \rightarrow \pi$ for the set of constraints $\{ (\delta = v) \rightarrow c \mid c \in \pi \}$.

$$\begin{aligned}
\pi_{\mathcal{S}l}(\mathbf{x}) &= \mathcal{V}(l, \mathbf{x}) \\
\pi_{\mathcal{S}\varphi_1 \vee \varphi_2}(\mathbf{x}) &= (\delta = 1) \Rightarrow \pi_{\mathcal{S}\varphi_1}(\mathbf{x}) \cup (\delta = 0) \Rightarrow \pi_{\mathcal{S}\varphi_2}(\mathbf{x}) \\
\pi_{\mathcal{S}\varphi_1 \wedge \varphi_2}(\mathbf{x}) &= \pi_{\mathcal{S}\varphi_1}(\mathbf{x}) \cup \pi_{\mathcal{S}\varphi_2}(\mathbf{x}) \\
\pi_{\mathcal{S}[\langle \Gamma \rangle]X\varphi}(\mathbf{x}) &= \bigcup_{i=1}^{|\text{Ind}_{\Gamma}|} (\delta_i = 1) \Rightarrow \left(\bigcup_{j=1}^{|\text{Ind}_{\overline{\Gamma}}|} C_{\Sigma_{\Gamma}^i \cup \Sigma_{\overline{\Gamma}}^j}(\mathbf{x}, \mathbf{y}_j) \right) \cup \sum_{i=1}^{|\text{Ind}_{\Gamma}|} \delta_i = 1 \cup \bigcup_{j=1}^{|\text{Ind}_{\overline{\Gamma}}|} \pi_{\mathcal{S}\varphi}(\mathbf{y}_j) \\
\pi_{\mathcal{S}[\llbracket \Gamma \rrbracket]X\varphi}(\mathbf{x}) &= \bigcup_{i=1}^{|\text{Ind}_{\Gamma}|} \left(\bigcup_{j=1}^{|\text{Ind}_{\overline{\Gamma}}|} (\delta_{ij} = 1) \Rightarrow C_{\Sigma_{\Gamma}^i \cup \Sigma_{\overline{\Gamma}}^j}(\mathbf{x}, \mathbf{y}_i) \cup \sum_{j=1}^{|\text{Ind}_{\overline{\Gamma}}|} \delta_{ij} = 1 \cup \pi_{\mathcal{S}\varphi}(\mathbf{y}_i) \right)
\end{aligned}$$

Figure 8: Monolithic encoding $\pi_{\mathcal{S}\varphi}$ for $\varphi \in \text{BATL}^*$, where the binary variables $\delta, \delta_i, \delta_{ij}$, the state variables $\mathbf{y}_j, \mathbf{y}_i$ and all auxiliary variables in $C_{\Sigma_{\Gamma}^i \cup \Sigma_{\overline{\Gamma}}^j}$ are fresh.

Without loss of generality, we assume below that for each agent index $a \in \{1, \dots, K, E\}$, there is a natural number b_a such that for each global state $q \in G$, the cardinality of $\text{prot}_a(q)$ is b_a (in practice, we can take b_a to be the maximum cardinality of $\text{prot}_a(q)$ for $q \in G$). We refer to b_a as the *action choice factor* (ACF) for agent a . We may assume that prot_a is given as b_a PWL functions $\text{prot}_a^1, \dots, \text{prot}_a^{b_a}$ mapping G to Act_a .

For a natural number n , we hereafter denote by $[n]$ the set $\{1, \dots, n\}$, and for a finite set S , denote by $|S|$ the cardinality of S . Given a set of agents $\Gamma \subseteq \text{Agt} \cup \{E\}$, denote by $\overline{\Gamma} = \text{Agt} \cup \{E\} \setminus \Gamma$ the *complement of Γ with respect to $\text{Agt} \cup \{E\}$* . Further, denote by Ind_{Γ} the Cartesian product of all $[b_a]$ with $a \in \Gamma$ and $\{0\}$ for $a \in \overline{\Gamma}$. Therefore, each element of Ind_{Γ} is a tuple $\sigma = (\sigma_1, \dots, \sigma_K, \sigma_E)$ of *action indices* that are non-negative integers, where $\sigma_a \in [b_a]$ if $a \in \Gamma$ and 0 if $a \in \overline{\Gamma}$. Ind_{Γ} represents *all distinct joint actions* of the agents in Γ and each element in Ind_{Γ} encodes one such joint action. As a consequence, $|\text{Ind}_{\Gamma}|$ is the number of all possible joint actions for the agents in Γ . We assume a natural (lexicographic) ordering of the elements of Ind_{Γ} , and denote by Σ_{Γ}^i the element number i in Ind_{Γ} . Given two tuples of integers σ^1 and σ^2 of the same length, denote by $\sigma^1 \cup \sigma^2$ the tuple σ such that $\sigma_a = \max(\sigma_a^1, \sigma_a^2)$.

Boldface letters $(\mathbf{x}, \mathbf{y}, \text{etc})$ denote tuples of real-valued MILP variables representing a global state and we call them *state variables*. Given a $(K+1)$ -tuple σ of natural numbers, $C_{\sigma}(\mathbf{x}, \mathbf{y})$ denotes the set of MILP constraints encoding the global transition

$$\mathbf{y} = \text{tr}(\mathbf{x}, \text{prot}_1^{\sigma_1}(\mathbf{x}), \dots, \text{prot}_K^{\sigma_K}(\mathbf{x}), \text{prot}_E^{\sigma_E}(\mathbf{x}));$$

this is admissible since tr and $\text{prot}_a^{\sigma_a}$ are PWL. For a literal l , $\mathcal{V}(l, \mathbf{x})$ denotes the set of MILP constraints over the state variables $\mathbf{x}$ that encode the set of global states at which l holds. For an atomic proposition p , $\mathcal{V}(p, \mathbf{x})$ expresses the set $V(p)$ and $\mathcal{V}(\neg p, \mathbf{x})$ expresses the set $G \setminus V(p)$ as MILP constraints on $\mathbf{x}$. We rely on $C_{\sigma}(\mathbf{x}, \mathbf{y})$ as building blocks for encoding the temporal evolution of the system and on $\mathcal{V}(l, \mathbf{x})$ for encoding the states of the system at which we stop the exploration. The former is formalised in the lemma below that follows from the fact that PWL functions can be represented by the feasible regions of MILPs.

Lemma 9. *Let $C_{\sigma}(\mathbf{x}, \mathbf{y})$ be a MILP program corresponding to*

$$\mathbf{y} = \text{tr}(\mathbf{x}, \text{prot}_1^{\sigma_1}(\mathbf{x}), \dots, \text{prot}_K^{\sigma_K}(\mathbf{x}), \text{prot}_E^{\sigma_E}(\mathbf{x})).$$

Given two states q and q' in $\mathcal{M}_{\mathcal{S}}$, we have that $q' = \text{tr}(q, \text{prot}_1^{\sigma_1}(q), \dots, \text{prot}_K^{\sigma_K}(q), \text{prot}_E^{\sigma_E}(q))$ iff there is an assignment $\mathbf{a}$ to $\text{vars}(C_{\sigma}(\mathbf{x}, \mathbf{y}))$ such that $s = \mathbf{a}(\mathbf{x})$, $s' = \mathbf{a}(\mathbf{y})$, and $\mathbf{a} \models C_{\sigma}(\mathbf{x}, \mathbf{y})$.

We present the encodings for the verification problem.

Distribution (A) Approved for public release; distribution is unlimited.

Algorithm 4 The MILP verification procedure.

```

1: procedure VERIFY( $\mathcal{S}, \varphi, \text{mono}$ )
2:   Input: NIS  $\mathcal{S} = (Agt, E, I, V)$ ; formula  $\varphi \in \text{bATL}^*$ 
3:   Output: True/False
4:    $\text{feasible} \leftarrow \text{False}$ 
5:    $\varphi' \leftarrow \text{NNF}(\neg\varphi \wedge p_I)$ 
6:   if  $\text{mono}$  then
7:      $\pi_{\mathcal{S}, \varphi'} \leftarrow$  MILP associated with  $\mathcal{S}$  and  $\varphi'$ 
8:      $\text{feasible} \leftarrow \text{MILP\_SOLVER}(\pi_{\mathcal{S}, \varphi'})$ 
9:   else
10:     $\Pi_{\mathcal{S}, \varphi'} \leftarrow$  Set of MILPs associated with  $\mathcal{S}$  and  $\varphi'$ 
11:    for  $\pi$  in  $\Pi_{\mathcal{S}, \varphi'}$  do
12:       $\text{aux} \leftarrow \text{MILP\_SOLVER}(\pi)$ 
13:      if  $\text{aux}$  is True then
14:         $\text{feasible} \leftarrow \text{True}$ 
15:        break
16:  return  $\neg\text{feasible}$ 

```

4.2.3 Monolithic Encoding.

We begin with an encoding which befits a verification procedure where only a single sequence of instructions is executed. Given a NIS $\mathcal{S}$ and a formula $\varphi \in \text{bATL}^*$, we construct a MILP $\pi_{\mathcal{S}, \varphi}$, whose feasibility corresponds to the satisfaction of φ by a state in $\mathcal{M}_{\mathcal{S}}$.

Definition 22. *The monolithic MILP encoding for a NIS $\mathcal{S}$ and a specification $\varphi \in \text{bATL}^*$, denoted $\pi_{\mathcal{S}, \varphi}$, is built inductively as the MILP $\pi_{\mathcal{S}, \varphi}(\mathbf{x})$, for a tuple of fresh state variables $\mathbf{x}$, according to the rules in Figure 8.*

The encoding of the literals in Figure 8 is $\mathcal{V}(l, \mathbf{x})$. The disjunction and conjunction are encoded similarly to the corresponding cases for $\text{bCTL}_{\mathbb{R}^{\leq}}$ in [1]. The encoding of the coalitional operators closely follows the semantics of alternating existential and universal quantification of strategies. Existential quantification is encoded via disjunctions, while universal quantifications via conjunctions. N -ary disjunction is captured by using N binary variables $\delta_1, \dots, \delta_N$ such that their sum amounts to 1 (and hence, only one can be true) and by employing indicator constraints enforcing that when δ_i is true, then the corresponding constraints must be satisfied. N -ary conjunction is captured by enforcing that all corresponding constraints are satisfied at the same time. The encoding of $\langle\langle \Gamma \rangle\rangle X$ results from generating $|\text{Ind}_{\bar{\Gamma}}|$ successor state variables $\mathbf{y}_j$ to ensure that for every strategy of the agents in $\bar{\Gamma}$, φ is satisfied at the corresponding successor. Symmetrically, the encoding of $[\![\Gamma]\!] X$ generates $|\text{Ind}_{\Gamma}|$ successor state variables $\mathbf{y}_i$.

It can be shown that the encoding is as intended.

Lemma 10. *Given a NIS $\mathcal{S}$, a formula $\varphi \in \text{bATL}^*$, and a state q in $\mathcal{M}_{\mathcal{S}}$ we have that $q \models \varphi$ iff there exists an assignment $\mathbf{a}$ to $\text{vars}(\pi_{\mathcal{S}, \varphi}(\mathbf{x}))$ such that $\mathbf{a} \models \pi_{\mathcal{S}, \varphi}(\mathbf{x})$ and $q = \mathbf{a}(\mathbf{x})$.*

Having defined the encoding, we now report the verification procedure in Algorithm 4 with mono set to **True**. It attempts to falsify $\mathcal{S} \models \varphi$ by searching for a state $q \in I$ such that $q \not\models \varphi$ (equivalently, $q \models \neg\varphi$). To this purpose, the formula passed to the monolithic encoding is $\neg\varphi \wedge p_I$ in negation normal form, where the negation is pushed down through to the atoms. Here, for simplicity, we assume that p_I is an additional atomic proposition in AP used for labelling the states in I , and that $V(p_I) = I$.

$$\begin{aligned}
\Pi_{\mathcal{S},l}(\mathbf{x}) &= \{[\mathcal{V}(l, \mathbf{x})]\}, & \Pi_{\mathcal{S},\langle\Gamma\rangle X\varphi}(\mathbf{x}) &= \bigcup_{i=1}^{|\text{Ind}_{\Gamma}|} \left(\bigtimes_{j=1}^{|\text{Ind}_{\overline{\Gamma}}|} \left\{ [C_{\Sigma_{\Gamma}^i \cup \Sigma_{\overline{\Gamma}}^j}(\mathbf{x}, \mathbf{y}_j)] \right\} \times \Pi_{\mathcal{S},\varphi}(\mathbf{y}_j) \right), \\
\Pi_{\mathcal{S},\varphi_1 \vee \varphi_2}(\mathbf{x}) &= \Pi_{\mathcal{S},\varphi_1}(\mathbf{x}) \cup \Pi_{\mathcal{S},\varphi_2}(\mathbf{x}), \\
\Pi_{\mathcal{S},\varphi_1 \wedge \varphi_2}(\mathbf{x}) &= \Pi_{\mathcal{S},\varphi_1}(\mathbf{x}) \times \Pi_{\mathcal{S},\varphi_2}(\mathbf{x}), & \Pi_{\mathcal{S},\llbracket\Gamma\rrbracket X\varphi}(\mathbf{x}) &= \bigtimes_{i=1}^{|\text{Ind}_{\Gamma}|} \left(\bigcup_{j=1}^{|\text{Ind}_{\overline{\Gamma}}|} \left\{ [C_{\Sigma_{\Gamma}^i \cup \Sigma_{\overline{\Gamma}}^j}(\mathbf{x}, \mathbf{y}_i)] \right\} \right) \times \Pi_{\mathcal{S},\varphi}(\mathbf{y}_i).
\end{aligned}$$

Figure 9: Compositional encoding $\Pi_{\mathcal{S},\varphi}$ for $\varphi \in \text{bATL}^*$.

4.2.4 Compositional Encoding.

To provide a more scalable verification procedure, we follow [1] by devising an encoding which creates separate MILP programs for each disjunctive case encountered in the monolithic encoding. Given a NIS $\mathcal{S}$ and a formula $\varphi \in \text{bATL}^*$, we define a set $\Pi_{\mathcal{S},\varphi}$ of MILP programs such that the satisfaction of φ by a state $q \in \mathcal{M}_{\mathcal{S}}$ corresponds to at least one of the programs $\pi \in \Pi_{\mathcal{S},\varphi}$ being feasible. Due to potentially multiple programs whose feasibility can be checked independently of each other, this encoding is particularly amenable to parallelisation.

We denote by $[C]$ the MILP program for the set C of linear constraints. Given sets $A = \{[A_1], \dots, [A_p]\}$ and $B = \{[B_1], \dots, [B_q]\}$ of MILP programs, we denote by $A \times B$ the *product* $\{[A_i \cup B_j] \mid i = 1, \dots, p, j = 1, \dots, q\}$.

Definition 23. *The compositional MILP encoding for a NIS $\mathcal{S}$ and a specification $\varphi \in \text{bATL}^*$, denoted $\Pi_{\mathcal{S},\varphi}$, is built inductively as the set of MILPs $\Pi_{\mathcal{S},\varphi}(\mathbf{x})$, for a tuple of fresh state variables $\mathbf{x}$, according to the rules in Figure 9.*

The compositional encoding operates on sets of MILPs. The encoding of the literals is straightforward and generates a singleton set containing the program corresponding to the literal. Disjunctions are dealt with by generating a different set of programs for each of the cases, and hence are implemented via the *union* operation over sets of programs. Conversely, conjunctions are encoded by taking all combinations of programs for each of the cases; it is implemented via the *product* operation over sets of programs. Therefore, the patterns of the form $\bigcup_i (\delta_i = 1) \Rightarrow C_i \cup \sum_i \delta_i = 1$ found in the monolithic encoding are replaced by $\bigcup_i \{[C_i]\}$, while “ $\cup$ ” found in the monolithic encoding is replaced by “ $\times$ ”.

We can show this encoding is correct.

Lemma 11. *Given a NIS $\mathcal{S}$, a formula $\varphi \in \text{bATL}^*$, and a state q in $\mathcal{M}_{\mathcal{S}}$, we have that $q \models \varphi$ iff there is a MILP $\pi(\mathbf{x}) \in \Pi_{\mathcal{S},\varphi}(\mathbf{x})$ and an assignment $\mathbf{a}$ to $\text{vars}(\pi(\mathbf{x}))$ such that $q = \mathbf{a}(\mathbf{x})$ and $\mathbf{a} \models \pi(\mathbf{x})$.*

The verification procedure of Algorithm 4 can also use the compositional encoding above, when *mono* is **False**. Notably, checking feasibility of the generated programs (lines 11–15) can be done sequentially, one after another, or in parallel across multiple processors. The theorem below states the correctness of the verification procedure for both the monolithic and compositional encodings.

Theorem 4. *Given a NIS $\mathcal{S}$ and a formula $\varphi \in \text{bATL}^*$, Algorithm 4 returns **False** iff $\mathcal{S} \not\models \varphi$.*

4.3 Experiments

In this section we present an implementation of the verification procedures described in Section 4.2 in a toolkit called VENMAS [2] which we evaluated on an air-traffic collision avoidance system.

VENMAS takes as input a NIS $\mathcal{S}$ and a BATL* specification φ . The observation function for each agent is given as linear combination of FFNNs. The local protocol and transition functions for each agent are given as PWL functions. The set I of initial states is given in the form of a hyper-rectangle $[l_1, u_1] \times \dots \times [l_\mu, u_\mu]$. The user can specify a parameter determining whether the monolithic or compositional procedure with parallel or sequential execution is to be used. The toolkit returns whether or not the specification φ holds on $\mathcal{S}$. We used Python to implement the tool and relied on Gurobi ver. 9.0 [13] as a back-end to resolve the feasibility of the generated MILPs.

4.3.1 Two-Agent Aircraft Collision Avoidance Scenario.

We validated the toolkit by using a two-agent extension of the scenario introduced in [15, 16] and studied in [1], VerticalCAS. We consider both aircraft to be controlled by their own independent VerticalCAS systems, removing the assumption that the intruder follows a horizontal trajectory; the intruder can now climb and descend independently to the ownship. This makes for a more complex scenario with a higher degree of branching in the resulting temporal evolution of the system. For simplicity we hereafter refer to this novel two-agent extension of our construction as VCAS[2].

We describe the global states of the scenario using the set of tuples $G = (h, \dot{h}_{\text{own}}, \dot{h}_{\text{int}}, \tau, ad_{\text{own}}, ad_{\text{int}}) \in [-3000, 3000] \times [-2500, 2500] \times [-2500, 2500] \times [0, 40] \times [9] \times [9]$, where h (ft) is the intruder altitude relative to the ownship, $\dot{h}_{\text{own}}$ (ft/sec) is the ownship's vertical climb rate, $\dot{h}_{\text{int}}$ (ft/sec) is the intruder's vertical climb rate, τ (secs) is the time to loss of horizontal separation of the aircraft, ad_{own} is the previous advisory issued to the ownship, and ad_{int} is the previous advisory issued to the intruder.

Each VerticalCAS system is composed of nine ReLU-FFNNs $F = \{(f_i : \mathbb{R}^4 \rightarrow \mathbb{R}^9) : i \in [9]\}$, one for each previously-issued advisory with four inputs, seven hidden layers of 45 nodes and nine outputs representing the score of each possible advisory.

4.3.2 Encoding VCAS[2] as a Neural Interpreted System.

Here we model VCAS[2] as a NIS. First, the environment states ℓ_E are tuples $(h, \dot{h}_{\text{own}}, \dot{h}_{\text{int}}, \tau)$, hence $L_E = [-3000, 3000] \times [-2500, 2500] \times [-2500, 2500] \times [0, 40]$. There are two agents $a \in \{\text{own}, \text{int}\}$ defined almost identically. Formally their components are as follows:

- local states ℓ_a are tuples $(\dot{h}_a, ad_a)$, where the current climb rate $\dot{h}_a$ constitutes the private state and the previously issued advisory ad_a corresponds to the percept, so $L_a = [-2500, 2500] \times [9]$,
- actions correspond to accelerations; we set $Act_a = \{0, \pm 3.0, \pm 7.33, \pm 9.33, \pm 9.7, \pm 11.7\}$,
- the protocol function returns the available accelerations, $prot_a(\ell_a) = \text{accs}(\text{compliant}(\ell_a), ad_a)$,
- for accelerations $\ddot{h}_a \in Act_a$, we define the local transition $tr_a((\dot{h}_a, ad_a), \ddot{h}_a, \cdot, \cdot) = \dot{h}_a + \ddot{h}_a \Delta\tau = \dot{h}'_a$,
- the observation function perceives the environment by computing the advisory according to VerticalCAS and information stored in the intermediate agent's local state $\ell'_a = (\dot{h}'_a, ad_a)$ and updated environment's local state ℓ'_E , so $obs_a(\ell'_a, \ell'_E) = ad'_a = \arg \max(\text{apply}_a(f_{ad_a}, \ell'_E))$,

where $\text{compliant}(\ell_a)$ is **True** iff $\dot{h}_a$ is compliant with ad_a ; $\text{accs}(\beta, ad_a) \subseteq 2^{Act_a}$ is a set of accelerations such that $\text{accs}(\text{True}, \cdot) = \{0\}$ and $\text{accs}(\text{False}, ad_a)$ is the set of permitted accelerations for the advisory ad_a ; $\Delta\tau = 1$; $\text{apply}_a : F \times L_E \rightarrow \mathbb{R}^9$ computes the output of a neural network on an environment state, such that $\text{apply}_{\text{own}}(f, \ell_E) = f(h, \dot{h}_{\text{own}}, \dot{h}_{\text{int}}, \tau)$ and $\text{apply}_{\text{int}}(f, \ell_E) = f(-h, \dot{h}_{\text{int}}, \dot{h}_{\text{own}}, \tau)$; $\arg \max : \mathbb{R}^9 \rightarrow [9]$ gives the index of the largest component.

Finally, the environment has one dummy action ε , giving $\alpha_E = \{\varepsilon\}$ and protocol function

	COMP-PAR				COMP-SEQ				MONOLITHIC			
h_I	-30	10	50	90	-30	10	50	90	-30	10	50	90
φ_1^1	0.33	0.39	0.34	0.48	0.32	0.34	0.32	0.97	0.04	0.08	0.05	0.04
φ_1^2	1.67	4.07	13.63	13.88	1.29	3.16	41.82	42.64	0.53	4.17	0.16	0.16
φ_1^3	104.29	—	—	—	59.48	—	—	—	303.35	994.45	0.67	0.85
φ_1^4	—	—	—	—	—	—	—	—	—	—	3.20	3.40
φ_1^5	—	—	—	—	—	—	—	—	—	—	9.63	9.22
φ_2^1	0.35	0.37	0.35	0.37	0.32	0.37	0.32	0.57	0.04	0.08	0.05	0.04
φ_2^2	1.23	3.07	2.21	2.47	1.17	2.99	5.66	5.76	0.52	4.25	0.16	0.16
φ_2^3	55.20	564.85	69.38	100.93	22.14	1823.65	213.81	309.56	906.68	933.99	0.66	0.84
φ_2^4	—	—	—	—	—	—	—	—	—	—	3.21	3.38
φ_2^5	—	—	—	—	—	—	—	—	—	—	9.51	9.23

Table 6: Verification times for VCAS[2] against properties φ_1^k and φ_2^k for various k and h_I . Grey cells indicate a **False** result, otherwise a **True** result was obtained. Dashes indicate a one hour timeout.

$prot_E(\ell_E) = \{\varepsilon\}$ for all $\ell_E \in L_E$, and the transition function is defined as

$$tr_E \left(\begin{bmatrix} h \\ \dot{h}_{own} \\ \dot{h}_{int} \\ \tau \end{bmatrix}, \ddot{h}_{own}, \ddot{h}_{int}, \varepsilon \right) = \begin{bmatrix} h' \\ \dot{h}_{own} + \ddot{h}_{own} \Delta \tau \\ \dot{h}_{int} + \ddot{h}_{int} \Delta \tau \\ \tau - \Delta \tau \end{bmatrix},$$

where $h' = h - \Delta \tau (\dot{h}_{own} - \dot{h}_{int}) - 0.5 \Delta \tau^2 (\ddot{h}_{own} - \ddot{h}_{int})$.

Given $Agt = \{own, int\}$ and atomic propositions $AP = \{NMAC, safe\}$, define the valuation function $V : AP \rightarrow 2^G$ as $V(NMAC) = \{q \in G \mid |h| \leq 100\}$ and $V(safe) = \{q \in G \mid h > 100 \text{ or } h < -100\}$ for global states $q = (h, \cdot, \cdot, \tau, \cdot, \cdot)$ with τ never close to zero.

We take the set of initial states $I = [h_I - 2, h_I + 2] \times \{-5.0\} \times \{5.0\} \times \{25.0\} \times \{1\} \times \{1\}$, where $h_I \in \{-30, 10, 50, 90\}$ ft. These represent potentially dangerous encounters with the aircraft inside the NMAC region with a descending ownship and an ascending intruder. When $h_I > 0$, the ownship is below the intruder, and when $h_I < 0$, the ownship is above the intruder. We fix an action choice factor of 2.

We can now construct a Neural Interpreted System for VCAS[2] for our two agents $a \in Agt$ as defined above.

4.3.3 Experimental Results.

We used VENMAS to evaluate VCAS[2] against the two specifications: $\varphi_1^k = \langle\langle own \rangle\rangle X^k \text{ safe}$, expressing the fact that the ownship has a strategy for the system to be in a safe configuration

after k steps, and $\varphi_2^k = \llbracket int \rrbracket X^k \text{ safe}$, expressing the fact that the intruder has no strategy to avoid the system to enter a safe configuration after k steps (note that φ_1^k implies φ_2^k but not the other way around). We evaluated the formulas with the set of initial states I defined

as above. All results were obtained on a machine with an Intel Core i7-7700K CPU, 16GB RAM, running a 64-bit version of Ubuntu 20.04. We denote by MONOLITHIC the results for the monolithic procedure and by COMP-PAR and COMP-SEQ those for the compositional procedure with parallel and sequential execution respectively.

In Table 6, we report the performance of VENMAS in terms of the amount of time (in seconds) to determine the truth of the specifications φ_1^k and φ_2^k for $k \in \{1, \dots, 5\}$ with initial

Distribution (A) Approved for public release; distribution is unlimited.

relative positions $h_I \in \{-30, 10, 50, 90\}$ for each execution mode. In all cases we used a timeout of one hour. Note that the choice of h_I has an impact on the truth of the various specifications.

Regarding the performance of the different encodings, we find that the monolithic encoding is more efficient than the compositional encoding for $h_I = 50$ and $h_I = 90$. We believe this to be because checking the infeasibility of a single MILP with tight bounds can be determined faster than checking several MILPs. For $h_I = -30$ and $h_I = 10$ the compositional encodings were more efficient. We believe this to be because if a single feasible MILP (from the set of generated MILP instances) is found, then a result can be returned immediately. It remains to be explored which encoding is preferable in general, as the difficulty of solving a MILP is hard to determine in advance. This would open the way to the development of heuristics to guide the choice of the encoding more likely to efficiently solve the problem under analysis.

We are unable to present a comparison with other tools because we are not aware of other tools supporting systems of multiple neural-symbolic agents and strategic properties as we do here. Similarly to [1], real values in VENMAS are represented by double-precision floating point numbers and we use the default constraint-satisfaction tolerance level of 10^{-6} in Gurobi.

5 Conclusions

The deliverable reported several advancements on the reachability analysis for feed-forward ReLU LE-CPSs.

First, in the context of verification of standalone feed-forward ReLU networks, we introduced the concept of dependency analysis which we developed in the context of a MILP-based verification method. The method further benefits from input splitting and symbolic interval propagation. We derived algorithms based on the resulting theory and reported the results obtained with *Venus*, a novel tool for the verification of neural networks. As we demonstrated experimentally on three different, widely used benchmarks, *Venus* could solve more verification queries than the present state-of-the-art tool based on complete methods. *Venus* is also the fastest tool to verify the correctness of a network; in some cases *Neurify* proved to be faster in finding counterexamples. In future work we intend to apply *Venus* to the verification of more complex specifications for neural networks including transformational robustness [20].

Second, we tackled verification of systems composed of a neural agent driven by deep feed-forward neural networks interacting with a non-deterministic environment. Such systems display branching evolutions. While the verification 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 FROZENLAKE and VerticalCAS scenarios. As demonstrated, while the parallel algorithm remains complete, it offers considerable advantages over its sequential counterpart when searching for witnesses to bounded reachability specifications in concrete examples.

Finally, we put forward the concept of neural interpreted systems to capture agents consisting of machine learning components for perception tasks and traditional logic-based components for decision making and control. The proposed architecture closely mirrors several forthcoming applications in which vision is realised via machine learning systems whereas decision making is executed via control logic, e.g., platoons of autonomous vehicles. We have formulated the verification problem against a language representing strategic interplay. Differently from traditional approaches in the literature, we have here opted for a bounded version of ATL with the aim of looking for shallow bugs. We have solved the resulting verification problem in terms of MILP and presented an encoding which, as we showed, is amenable to parallelisation.

We reported on a prototype that we developed implementing both encodings and evaluated

against a sophisticated model from the avionics literature. The evaluation demonstrated the correctness of the approach and that small systems can feasibly be verified. It also highlighted, however, that a challenge of the present method lies in its scalability, as the MILPs become more difficult to solve as the number of non-linearities encountered in the encodings grow rapidly with the number of steps considered. This problem is exacerbated by the highly branching nature of the models often considered in a strategic setting.

In future, we plan further work on conquering the scalability of the verification problems, both in the case of standalone neural networks and in the case of closed-loop systems. In particular, we are going to explore abstraction approaches in order to achieve gains of 1-2 orders of magnitude.

References

- [1] M. E. Akintunde, E. Botoeva, P. Kouvaros, and A. Lomuscio. Formal verification of neural agents in non-deterministic environments. In *Proceedings of the 19th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS20)*, pages 25–33. IFAAMAS, 2020.
- [2] M. E. Akintunde, E. Botoeva, P. Kouvaros, and A. Lomuscio. VENMAS: Verification of neural-symbolic multi-agent systems, <https://vas.doc.ic.ac.uk/software/neural/>, 2020.
- [3] M. E. Akintunde, A. Lomuscio, L. Maganti, and E. Pirovano. Reachability analysis for neural agent-environment systems. In *Proceedings of the 16th International Conference on Principles of Knowledge Representation and Reasoning (KR18)*, pages 184–193. AAAI Press, 2018.
- [4] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):672–713, 2002.
- [5] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.
- [6] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
- [7] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
- [8] C. D’Ambrosio, A. Lodi, and S. Martello. Piecewise linear approximation of functions of two variables in MILP models. *Operations Research Letters*, 38(1):39–46, 2010.
- [9] T. Dreossi, A. Donzé, and S. Seshia. Compositional falsification of cyber-physical systems with machine learning components. *Journal of Automated Reasoning*, 63(4):1031–1053, 2019.
- [10] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. *Real-Time Systems*, 4(4):331–352, 1992.
- [11] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about knowledge*. MIT Press, Cambridge, 1995.

- [12] A. Garcez, L. Lamb, and D. Gabbay. *Neural-symbolic cognitive reasoning*. Cognitive Technologies. Springer, 2009.
- [13] Z. Gu, E. Rothberg, and R. Bixby. Gurobi optimizer reference manual. <http://www.gurobi.com>, 2020. [Accessed: 2020-07-02].
- [14] K. Julian, J. Lopez, J. Brush, M. Owen, and M. Kochenderfer. Policy compression for aircraft collision avoidance systems. In *Proceedings of the 35th Digital Avionics Systems Conference (DASC16)*, pages 1–10, 2016.
- [15] K. D. Julian and M. J. Kochenderfer. A reachability method for verifying dynamical systems with deep neural network controllers. *CoRR*, abs/1903.00520, 2019.
- [16] K. D. Julian, S. Sharma, J-B. Jeannin, and M. J. Kochenderfer. Verifying aircraft collision avoidance neural networks through linear approximations of safe regions. *CoRR*, abs/1903.00762, 2019.
- [17] N. Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing (STOC84)*, pages 302–311. ACM, 1984.
- [18] G. Katz, C. W. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *Proceedings of the 29th International Conference on Computer Aided Verification (CAV17)*, volume 10426 of *Lecture Notes in Computer Science*, pages 97–117. Springer, 2017.
- [19] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljic, D. L. Dill, M. J. Kochenderfer, and C. W. Barrett. The Marabou framework for verification and analysis of deep neural networks. In *Proceedings of the 31st International Conference on Computer Aided Verification (CAV19)*, pages 443–452, 2019.
- [20] P. Kouvaros and A. Lomuscio. Formal verification of cnn-based perception systems. *arXiv preprint arXiv:1811.11373*.
- [21] A. Krizhevsky, V. Nair, and G. Hinton. The CIFAR-10 dataset. <http://www.cs.toronto.edu/kriz/cifar.html>, 2014.
- [22] Y. LeCun, C. Cortes, and C. J. Burges. The MNIST database of handwritten digits, 1998.
- [23] A. Lomuscio, H. Qu, and F. Raimondi. MCMAS: A model checker for the verification of multi-agent systems. *Software Tools for Technology Transfer*, 19(1):9–30, 2017.
- [24] P. Maes. Modeling adaptive autonomous agents. *Artificial life*, 1(1–2):135–162, 1993.
- [25] OpenAI. Frozenlake-v0. <https://gym.openai.com/envs/FrozenLake-v0/>, 2019.
- [26] W. Penczek, B. Woźna, and A. Zbrzezny. Bounded model checking for the universal fragment of CTL. *Fundamenta Informaticae*, 51(1-2):135–156, 2002.
- [27] H. Siegelmann and E. Sontag. On the computational power of neural nets. *Journal of Computer and System Sciences*, 50(1):132–150, 1995.
- [28] R. S. Sutton and A. G. Barto. *Reinforcement Learning – An Introduction*. MIT Press, 1998.

- [29] VENMAS. VERification of Neural Multi-Agent Systems <https://vas.doc.ic.ac.uk/software/neural>, 2020.
- [30] Venus. <https://vas.doc.ic.ac.uk/software/neural>, 2019.
- [31] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Efficient formal safety analysis of neural networks. In *Proceedings of the 31st Annual Conference on Neural Information Processing Systems 2018 (NeurIPS18)*, pages 6369–6379, 2018.
- [32] M. Wooldridge and A. Lomuscio. Reasoning about visibility, perception, and knowledge. In *International Workshop on Agent Theories, Architectures, and Languages*, pages 1–12. Springer, 1999.

Functional evaluation

Imperial College London

1 Introduction

Neural Networks (NN) have achieved impressive breakthroughs across several domains of science – see [20] for a comprehensive catalogue of success stories. Although being versatile and efficient, neural networks are known to be susceptible to adversarial perturbations [10, 27], i.e., imperceptible modifications to their inputs can lead to unexpected, and often inexplicable, consequences on the outputs produced by the network. This lack of reliability and transparency has hindered a wider adoption of neural networks in safety and security-critical settings.

Verification of neural networks (VNN) offers a promising solution to alleviate this problem. Since its inception, the field of VNN has experienced an impressive growth, with several methods and tools being developed for different classes of neural architectures and specifications [29]. In this report, we consider two neural network-based industrial systems developed by Boeing for the DARPA Assured Autonomy programme and present results pertaining to their formal verification. Specifically, we have improved some functionalities of the VENUS verification toolkit [6], and used it to analyse

1. an object detection system trained for open category detection and
2. a neural controller trained to assist landing in non-towered airports.

In particular VENUS is used to determine conditions under which the systems can operate safely or generate counterexamples that comprehensively show the circumstances under which safety cannot be guaranteed.

Related work. In recent years, there has been a large amount of work on reasoning on safety of neural networks. Formal verification for neural networks concerns checking whether a network satisfies an input/output specification defining the set of possible inputs and the set of admissible outputs, respectively.

Several efficient methods exist to check this property for different classes of neural architectures, e.g., [2, 4, 5, 6, 12, 17, 24, 26, 28]. While an exhaustive

review of existing approaches is outside the scope of this report, we remark that only some methods are *complete*, i.e., if a counterexample exists, these approaches are guaranteed to find it. In contrast, *incomplete* approaches are normally based on various forms of convex relaxations of the network, and they can be conservative and thus fail to find some counterexamples. Despite considerable achievements of the cited works, the verification problem remains challenging for industry-scale networks. Indeed, the problem is known to be NP-complete [16], thereby motivating a very active area of research to improve the scalability of neural network verification.

An also recent body of work concerns the verification of closed-loop and cyber-physical systems equipped with learned controllers – see, e.g., [30, 14, 15, 1]. Verification at this level typically entails checking richer specifications pertaining to the systems’ temporal evolutions. Since verification over specifications expressed in standard temporal logics, such as LTL and CTL [13], is often undecidable [1], verification procedures in this context are typically restricted to checking *bounded* specifications which express properties concerning only a bounded number of execution steps.

2 Network verification with the Venus toolkit

VENUS is a state-of-the-art sound and complete verification toolkit for ReLU-based feed-forward neural networks, developed by the Imperial College London team. In this section, we give a formal description of the verification problem tackled by VENUS and an outline of the procedure in VENUS to solve it. We begin with the definition of feed-forward neural networks.

Feed-forward ReLU networks. A *feed-forward neural network* (FFNN)

is a vector-valued function $\mathbf{f}: \mathbb{R}^m \rightarrow \mathbb{R}^n$ that composes a sequence of $k > 1$ *layers*, where each layer $\mathbf{f}^{(i)}$ is the composition of an affine transformation and a non-linear *activation* function. That is, $\mathbf{f}^{(i)}(\mathbf{x}^{(i-1)}) \triangleq \text{act}^{(i)}(\mathbf{W}^{(i)}\mathbf{x}^{(i-1)} + \mathbf{b}^{(i)})$, where $\mathbf{x}^{(i-1)}$, $i > 0$, is the input to the i -th layer, $\mathbf{x}^{(0)}$ is the input to the network, $\text{act}^{(i)}$ is the activation function of the i -th layer, and $\mathbf{z}^{(i)} = \mathbf{W}^{(i)}\mathbf{x}^{(i-1)} + \mathbf{b}^{(i)}$ is the vector of *pre-activations* of the layer which is the affine transformation of the previous layer’s output for a weight matrix $\mathbf{W}^{(i)}$ and a bias vector $\mathbf{b}^{(i)}$. VENUS focuses on the Rectified Linear Unit (ReLU) activation function, $\text{ReLU}(\mathbf{z}^{(i)}) \triangleq \max(0, \mathbf{z}^{(i)})$, $i > 0$, where the maximum function is applied element-wise on $\mathbf{z}^{(i)}$.

Verification problem. Given a feed-forward neural network, VENUS answers positively or negatively as to whether the output of the network for every input within a linearly definable set ¹ of inputs is contained within a linearly definable set of outputs. Formally, we have:

¹A linearly definable set is a set that can be expressed as a finite set of linear constraints over real-valued and integer variables.

Definition 2.1 (Verification problem) *Given a FFNN $f: \mathbb{R}^m \rightarrow \mathbb{R}^n$, a linearly definable set of inputs $\mathcal{X} \subset \mathbb{R}^m$ and a linearly definable set of outputs $\mathcal{Y} \subset \mathbb{R}^n$, the verification problem concerns determining whether $\forall \mathbf{x} \in \mathcal{X}: \mathbf{f}(\mathbf{x}) \in \mathcal{Y}$.*

Among the many instantiations of the above verification problem, the *local adversarial robustness problem* is perhaps the most studied [25, 16]. Answering the local adversarial robustness problem requires establishing whether all images within a norm-ball of a given image are classified equivalently by f , and can be instantiated by setting $\mathcal{X} = \{\mathbf{x}' \mid \|\mathbf{x} - \mathbf{x}'\|_p \leq \epsilon\}$ and $\mathcal{Y} = \{\mathbf{y} \mid \forall i \neq c: y_i < y_c\}$ for a given image $\mathbf{x}$ with class label c , perturbation radius $\epsilon \geq 0$ and norm $\|\cdot\|_p$.

Verification with Venus. VENUS implements an MILP-based verification method whereby the verification problem is translated into a Mixed Integer Linear Program (MILP). An MILP is an optimisation problem whereby an objective function is sought to be maximised subject to a set of linear constraints over real-valued and integer variables. The feasibility status of the MILP program associated with a verification problem, i.e., whether there is an assignment to the variables of the MILP that satisfies all constraints, has a strict correspondence to the satisfaction of the verification problem: the verification problem is satisfied if and only if its MILP program has no feasible solution. VENUS makes use of the big-M method [23, 3] to translate the verification problem into MILP and relies on GUROBI [11] as the MILP solving back-end. In addition, VENUS implements a number of methods that aim at reducing the search space of feasible solutions that needs to be considered by GUROBI, including dependency analysis and input domain splitting [6].

3 Neural network-based systems for the aircraft domain

In this section, we present our main results pertaining to the verification of two neural network-based industrial systems developed by Boeing. First we will focus on verifying the robustness of a high-dimensional image classifier for open object detection. Then, we will consider the verification of a landing assistant for non-towered airports.

3.1 Object detection with open categories

Problem description. In a real-world application, a neural network classifier is very likely to encounter novel obstacles (i.e., open categories), such as novel types of ground vehicles and novel static objects. It is desirable that the classifier is able to detect these novel objects without previous training on the objects thereof while producing only a small number of false alarms, i.e., an alarm on the detection of a novel object even though the object is not novel. The classifier should also have a certain robustness against adversarial perturbations on the image input. The following study is concerned with providing risk-bounded



Figure 1: Sample images in the data set for open-category detection.

assurance against open category instances for an image classification system developed by Boeing.

Data set and neural network classifier. For this problem, Boeing has created a labelled data set that includes contiguous image sequences as aircraft approach novel obstacles or conditions. In particular, the data set contains three classes of objects: *ground vehicle*, *person*, and *airplane*. Fig. 1 shows three sample images in each category. Each image has three channels, with 255×255 pixels in each channel. The image input is thus expressed as a three-dimensional matrix with a high dimension $255 \times 255 \times 3$. Boeing has also trained a large convolutional neural network (CNN) with roughly 10^5 neurons using two categories (vehicle and person) as the training set. The other category (i.e., airplane) is used as open objects for the neural network to test its performance of detecting novel objects.

The architecture of this large CNN model has two convolutional layers of 16 and 32 filters respectively (size 5×5 and 3×3 , respectively), followed by one global average pooling layer, two fully connected layers with 128 and 2 neurons respectively, and one final softmax layer. The softmax layer normalizes the output into the interval $[0, 1]$ such that its outputs add up to one. The activation function in each layer is the standard ReLU operator $\hat{z} = \max\{z, 0\}$. Therefore, the CNN model maps an image input $x \in \mathbb{R}^{255 \times 255 \times 3}$ to an output $y \in \mathbb{R}^2$ and classifies x using the following logic: if $y_1 \geq t_1$, it outputs the label “*vehicle*” with confidence y_1 ; if $y_2 \geq t_2$, it outputs the label “*person*” with confidence y_2 ; otherwise, it reports the input image as a novel object. The thresholds t_1 and t_2 were carefully chosen to balance the false alarm rate of (20%) and the overall prediction error rate (30%).

Robustness verification setup. We aim to verify the local robustness of the above CNN against adversarial perturbations on the image input. Formally, given a correctly classified image x_0 (person, vehicle, or open object) and a perturbation radius $\epsilon > 0$, we would like to certify

$$\text{for all } x \text{ s.t. } \|x - x_0\|_\infty \leq \epsilon \text{ we have } f(x) = f(x_0), \quad (1)$$

or to identify a counterexample $\hat{x}$ to falsify the condition above. In (1), $f(\cdot)$ denotes the CNN model. If (1) is true, then the neural network is certified to

be locally robust for the image input x_0 against any adversarial perturbation up to ϵ . Otherwise, a counterexample can be found which can be used to augment the training set to improve the robustness of the model. Note that (1) is a particular instantiation of the verification problem in Section 2.

Extension of Venus and numerical results: The images in this case study have a very high dimension, i.e., $x \in \mathbb{R}^{255 \times 255 \times 3}$, similar to the dimension of those in Imagenet [9]. We note that most existing complete robustness verifiers only test their verification performance on standard benchmarks, such as the MNIST dataset [19] or the CIFAR10 dataset [18], where the images have a much lower dimension ($28 \times 28 \times 1$ and $32 \times 32 \times 3$, respectively). In addition, the CNN model is large (approximately 10^5 neurons) as compared to the standard models considered in neural network verification. As a consequence, VENUS could not efficiently generate the MILP encoding of the robustness verification problem (1), either because of memory issues or because of exceeding the time consumption limit (see below). To overcome this, we have improved VENUS’s implementation by exploiting the special structure of CNNs, i.e., that the value of a neuron depends only on a small subset of the neurons in the previous layers, to obtain a much more scalable MILP encoding method.

For the numerical experiments, we used a machine with an Intel Core i9 9900X 3.5 GHz 10-core CPU, 128 GB RAM running Fedora 30 with Linux kernel 5.3. We randomly selected 20 images from the data set that are correctly classified by the CNN model. We then used the extended version of VENUS to solve the robustness verification problem (1) with a timeout limit of 7200 seconds. To test the robustness of the CNN model, we varied the perturbation radius from 1.0×10^{-4} to 1.0×10^{-2} . Table 1 lists the results. We can see that the CNN model is robust with respect to the small perturbation (1.0×10^{-4}); in this case VENUS could return a positive certificate within 1 hour. Concerning the bigger perturbation radii, the CNN model becomes less robust; in this case VENUS could identify concrete counterexamples for 10 images out of 20 within 7200 seconds. In particular, VENUS identified a counterexample for the “vehicle” image shown in Fig 1(b) for $\epsilon = 10^{-2}$. This counterexample fooled the CNN to classify it an open object. Finally, we note that since VENUS is a complete verifier the robustness of the CNN can be certified w.r.t the rest of the images by increasing the timeout.

Radius ϵ	Robust		Non-robust		Time-out (7200 s)
	Number	Avg time (s)	Number	Avg time (s)	
1.0×10^{-4}	20	2500	0	—	0
1.0×10^{-3}	6	2720	2	4850	12
1.0×10^{-2}	0	—	10	4200	10

Table 1: Numerical results via VENUS on randomly selected 20 images

3.2 Collision avoidance for landing at non-towered airports

Problem description. Automated collision avoidance has become an essential component of every autonomous system. Whilst the safety of commercial aviation relies on centralized guidance from Air Traffic Control (ATC), private aviation must rely on onboard systems, especially when operating at non-towered airports. To prevent collisions, the regulator requires all vehicles to maintain a minimum safe horizontal separation which is advised via ATC commands, a set of rules defined in the pilot’s handbook and the airport-specific traffic pattern.

This case study poses the collision avoidance problem as a data-driven detection problem. The goal is to detect the collisions when approaching landing by analysing the dynamic state of the airplane (Agent) and another vehicle (Intruder). The detection is done via an ML-based inference model trained on the previous (simulated) experience. A *positive* detection triggers the rejection of the landing and makes the Agent go for another landing approach. This rejection guarantees safe separation in the vertical space.

Simulation environment. Boeing developed a lightweight Python simulation integrated with the OpenAI GYM framework [7] to facilitate the study of the problem described above. The surrogate environment simulates a 2D collision avoidance problem that mimics the real task (on the ground collision avoidance for the aircraft landing). The environment simulates the movements of multiple Intruder vehicles on a 10×10 kilometres square. The Agent has to either continue the automated landing or reject it to provide minimal horizontal separation. The environment incorporates a probabilistic behaviour model for the Intruder and the use of scripted configuration files: the former allows to randomize the trajectory of the Intruder vehicle and simulate a violation of the traffic rules; the latter enables quick reconfiguring of simulated scenarios.

The surrogate simulation imitates the dynamics of both the Agent and the Intruder vehicles selected for the experiment. All ground vehicles are represented by a simplistic dynamic model (mass-less Dubin’s car). Each vehicle uses a base class that takes control inputs in form of steering and acceleration commands in return for its updated position and speed. For the Agent vehicle, we simulate the dynamics of a single-engine turboprop Cessna 208 Grand Caravan. The aircraft control is different from 2D car-like dynamics and does not have conventional brakes, acceleration, and steering. The effectiveness of the controls depends on the aircraft’s altitude, airspeed, etc. A PID-based waypoint controller provides low-level control for the Agent and Intruders. The controller defines trajectories as lists of waypoints to be followed and adds a controlled level of noise to the controller state to make trajectories more natural.

Neural network classifier. The rejection action is commanded by a pre-trained neural network classification model that predicts the probability of collision between the Agent and Intruder vehicles projected on a 2D surface. The

network is fully-connected and is trained using data gathered from the simulator. Each data sample represents a state vector that captures the position, heading and speed of the Agent and the Intruder vehicles. Samples are labelled either as 0 or 1, depending on whether the simulation run from which they were gathered resulted in a collision or not. The neural network has 8 input neurons, 2 hidden layers with 128 nodes each and ReLU activations, and 1 output neuron that uses a Sigmoid activation to produce a continuous collision probability within 0 and 1.

To evaluate the performance of the system, Boeing developed a run-time rejection system that utilizes the neural network to predict the probability of collision and compares this with a predefined threshold λ to command the rejection. The runtime system evaluates the chance of collision with each Intruder vehicle on the ground individually, and triggers a rejection if the inference model predicts a probability greater than λ for any Intruder.

ϵ	Safe configurations				Collision configurations			
	Robust	Avg time (s)	Not robust	Avg time(s)	Robust	Avg time (s)	Not robust	Avg time(s)
0.05	50	0.09	0	-	48	0.11	2	0.33
0.1	50	0.09	0	-	48	0.10	2	0.38
0.2	50	0.08	0	-	48	0.10	2	0.39
0.5	50	0.08	0	-	48	0.11	2	0.39
1	50	0.08	0	-	47	0.10	3	0.33
1.5	50	0.07	0	-	45	0.11	5	0.37

Table 2: Robustness results for safe configurations and collision configurations.

Robustness analysis. We conducted a formal analysis of the neural classifier to assess its local robustness against adversarial perturbations. Formally, given an input x_0 and a perturbations radius $\epsilon > 0$, we check if for all x such that $\|x - x_0\|_\infty \leq \epsilon$ it holds that $f(x) \diamond \lambda^2$, where $f(\cdot)$ denotes the network trained by Boeing and $\diamond \in \{\geq, \leq\}$ depending on whether the initial input x_0 belonged to a collision trajectory or not. Intuitively, the inference model should always compute prediction probabilities that are $\geq \lambda$ (resp. $\leq \lambda$) for inputs x in the vicinity of an x_0 for which a collision happened (resp. did not happen).

We collected 50 samples from the simulator for which the neural network correctly predicted collision and 50 for which collisions did not occur. Table 2 reports our results for a rejection threshold of $\lambda = 0.75$. The inference model appears to be robust for perturbations up to $\epsilon = 1.5$ for inputs that belong to a safe trajectory, i.e., the model consistently predicted low collision probabilities for the perturbations considered. However decisions appear to be less robust in the other case; we hypothesise this is because of the fact that increasing ϵ may effectively create input configurations for which rejection commands should not be triggered. In all cases, VENUS was able to solve the verification problem within less than a second.

²Note that VENUS does not support the Sigmoid function used in the output layer; we therefore use its inverse to compute the preimage of λ and compare this value with the pre-activation value of the output node.

Analysing sensitivity to single inputs. Our previous analysis considered adversarial perturbations that are allowed to modify all input components at the same time. This corresponds to a scenario where all sensors of the Agent vehicle are subject to failure. However, more realistic failures may involve only a subset of the sensors; moreover, different sensor failures may have different consequences on the final prediction of the inference model.

To investigate this, we carry out a formal analysis where the inference model is verified against a variation of the local robustness property previously used. Namely, we apply adversarial ϵ -bounded perturbations to a single input component at the time and verify robustness in the resulting scenario. Similarly to the previous experiments, we sampled 50 instances for which collisions were flagged in the simulator and tested increasing perturbation radii. Table 3 reports sample results obtained for $\epsilon = 1$. Our anal-

Component	Not robust	Avg time (s)
x_E	1	0.25
y_E	1	0.26
θ_E	1	0.26
s_E	1	0.25
x_I	1	0.24
y_I	1	0.24
θ_I	1	0.26
s_I	3	0.29

Table 3: Sensitivity analysis results for perturbations applied to single input components and perturbation radius $\epsilon = 1$.

ysis helped us identify that predictions seem to be comparably less robust to perturbations applied to the input component related to the speed of the Intruder vehicle. This can be explained by the fact that changes in speed may create scenarios where the Intruder proceeds slowly and thus collisions can be avoided. Again, VENUS was able to solve all verification queries within less than a second.

4 Phase III: outlook

We considered two neural network-based industrial systems developed by Boeing and showed how formal verification can be used to provide assurance guarantees for said systems. The VENUS verification toolkit was successfully employed to formally analyse the behaviour of these models, providing proofs of safety or counterexamples to show when safety could not be guaranteed. Despite promising results, several challenges remain to be addressed in Phase III.

Scalability is admittedly the biggest challenge in this arena, as industrial applications often require checking neural models that contain hundreds of thousands parameters, if not millions. While the progress in Phase II by the Imperial College London team, as well as some other recent efforts in VNN, has contributed to significant scalability improvements [21, 22] – more work still needs to be done to address industrial-scale problems. This need becomes even more evident when verification is performed at system level, i.e., when the neural network-based system is considered in its entirety and closed-loop behaviours are analysed.

Another important direction where the expertise of the formal methods com-

munity may be determinant is that of formalising *richer specifications* beyond local adversarial robustness, especially in the case of perception systems such as the one studied in Section 3. Recent developments in VNN have addressed new specifications, such as robustness against semantic perturbations that can alter, e.g., saturation or contrast in an image [17], and geometric perturbations that apply transformations such as rotations to input images [5].

We conclude by highlighting that safe deployment of neural network-based systems will also require *runtime assurance methods* that are able to detect anomalous behaviour during execution. Different proposals have been made within the formal verification community, see, e.g., [8]; however several interesting open questions remain in this domain.

References

- [1] Akintunde, M., Botoeva, E., Kouvaros, P., Lomuscio, A.: 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 (2020)
- [2] Akintunde, M., Kevorchian, A., Lomuscio, A., Pirovano, E.: Verification of RNN-based neural agent-environment systems. In: Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI19). pp. 6006–6013. AAAI Press (2019)
- [3] Anderson, R., Huchette, J., Ma, W., Tjandraatmadja, C., Vielma, J.: Strong mixed-integer programming formulations for trained neural networks. *Mathematical Programming* pp. 1–37 (2020)
- [4] Bak, S., Tran, H., Hobbs, K., Johnson, T.: Improved geometric path enumeration for verifying relu neural networks. In: CAV20. Lecture Notes in Computer Science, vol. 12224, pp. 66–96. Springer (2020)
- [5] Balunovic, M., Baader, M., Singh, G., Gehr, T., Vechev, M.: Certifying geometric robustness of neural networks. In: NeurIPS19. pp. 15287–15297 (2019)
- [6] Botoeva, E., Kouvaros, P., Kronqvist, J., Lomuscio, A., Misener, R.: Efficient verification of neural networks via dependency analysis. In: Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI20). pp. 3291–3299. AAAI Press (2020)
- [7] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: Openai gym. arXiv preprint 1606.01540 (2016)
- [8] Cheng, C., Nührenberg, G., Yasuoka, H.: Runtime monitoring neuron activation patterns. In: Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE19). pp. 300–303. IEEE (2019)

- [9] Deng, J., Dong, W., Socher, R., Li, L., Li, K., Fei-Fei, L.: Imagenet: A large-scale hierarchical image database. In: Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR09). pp. 248–255. IEEE (2009)
- [10] Goodfellow, I., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. arXiv preprint arXiv:1412.6572 (2014)
- [11] Gu, Z., Rothberg, E., Bixby, R.: Gurobi optimizer reference manual. <http://www.gurobi.com> (2020)
- [12] Henriksen, P., Lomuscio, A.: 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 (2020)
- [13] Huth, M.A., Ryan, M.: Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge University Press (2000)
- [14] Ivanov, R., Carpenter, T., Weimer, J., Alur, R., Pappas, G., Lee, I.: Verifying the safety of autonomous systems with neural network controllers. *ACM Trans. Embed. Comput. Syst.* **20**(1), 7:1–7:26 (2021)
- [15] Ivanov, R., Weimer, J., Alur, R., Pappas, G., Lee, I.: 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 (2019)
- [16] Katz, G., Barrett, C., Dill, D., Julian, K., Kochenderfer, M.: 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 (2017)
- [17] Kouvaros, P., Lomuscio, A.: Formal verification of cnn-based perception systems. arXiv preprint arXiv:1811.11373 (2018)
- [18] Krizhevsky, A., Hinton, G.: Learning multiple layers of features from tiny images (2009)
- [19] LeCun, Y.: The mnist database of handwritten digits. <http://yann.lecun.com/exdb/mnist/> (1998)
- [20] LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. *Nature* **521**(7553), 436–444 (2015)
- [21] Li, L., Qi, X., Xie, T., Li, B.: Sok: Certified robustness for deep neural networks. arXiv preprint 2009.04131 (2020)
- [22] Liu, C., Arnon, T., Lazarus, C., Barrett, C., Kochenderfer, M.: Algorithms for verifying deep neural networks. arXiv preprint 1903.06758 (2019)

- [23] Lomuscio, A., Maganti, L.: An approach to reachability analysis for feed-forward relu neural networks. arXiv preprint 1706.07351 (2017)
- [24] Narodytska, N., Kasiviswanathan, S., Ryzhyk, L., Sagiv, M., Walsh, T.: Verifying properties of binarized deep neural networks. arXiv preprint arXiv:1709.06662 (2017)
- [25] Pulina, L., Tacchella, A.: An abstraction-refinement approach to verification of artificial neural networks. In: Proceedings of the 22nd International Conference on Computer Aided Verification (CAV10). Lecture Notes in Computer Science, vol. 6184, pp. 243–257. Springer (2010)
- [26] Singh, G., Gehr, T., Mirman, M., Püschel, M., Vechev, M.: Fast and effective robustness certification. In: Advances in Neural Information Processing Systems (NeurIPS18). pp. 10802–10813 (2018)
- [27] Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., Fergus, R.: Intriguing properties of neural networks. In: Proceedings of the 2nd International Conference on Learning Representations (ICLR14) (2014)
- [28] Tran, H., Yang, X., Lopez, D.M., Musau, P., Nguyen, L., Xiang, W., Bak, S., Johnson, T.: NNV: the neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. In: CAV20. Lecture Notes in Computer Science, vol. 12224, pp. 3–17. Springer (2020)
- [29] VNN-COMP: Verification of Neural Networks Competition (VNN-COMP20). <https://sites.google.com/view/vnn20/vnncomp> (2020), online; accessed 23 March 2021
- [30] Xiang, W., Tran, H., Rosenfeld, J., Johnson, T.: 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 (2018)

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

ACAS	Autonomous Collision Avoidance System
AI	Artificial Intelligence
ARCH	Applied Verification for Continuous and Hybrid Systems
ATL	Alternating-time temporal Logic
BNF	Backus–Naur Form
CIFAR	Canadian Institute for Advanced Research
CPS	Cyber-physical System
CPU	Central Processing Unit
CTL	Computation-tree Logic
DARPA	Defense Advanced Research Projects Agency
LEC	Learning-enabled Component
LE-CPS	Learning-enabled Cyber-physical System
LP	Linear Programming
LTL	Linear Temporal Logic
MAS	Multi-Agent System
MILP	Mixed-integer Linear Programming
ML	Machine Learning
MNIST	Modified National Institute of Standards and Technology
NN	Neural Networks
RAM	Random Access Memory
ReLU	Rectified Linear Unit
RLT	Reformulation-linearization Technique
SDP	Semi-definite Programming
SIP	Symbolic Interval Propagation
TA	Technical Area