Vehicle: Bridging the Embedding Gap in the Verification of Neuro-Symbolic Programs

Matthew L. Daggitt
University of Western Australia, Perth, Australia

Wen Kokke
Well-Typed, UK

Robert Atkey
University of Strathclyde, Glasgow, UK

Ekaterina Komendantskaya
Heriot-Watt and Southampton Universities, UK

Natalia Slusarz
Heriot-Watt University, Edinburgh, UK

Luca Arnaboldi
University of Birmingham, Birmingham, UK

Abstract

Neuro-symbolic programs, i.e. programs containing both machine learning components and traditional symbolic code, are becoming increasingly widespread. Finding a general methodology for verifying such programs is challenging due to both the number of different tools involved and the intricate interface between the "neural" and "symbolic" program components. In this paper we present a general decomposition of the neuro-symbolic verification problem into parts, and examine the problem of *the embedding gap* that occurs when one tries to combine proofs about the neural and symbolic components. To address this problem we then introduce VEHICLE– standing as an abbreviation for a "verification condition language" – an intermediate programming language interface between machine learning frameworks, automated theorem provers, and dependently-typed formalisations of neuro-symbolic programs. VEHICLE allows users to specify the properties of the neural components of neuro-symbolic programs once, and then safely compile the specification to each interface using a tailored typing and compilation procedure. We give a high-level overview of VEHICLE's overall design, its interfaces and compilation & type-checking procedures, and then demonstrate its utility by formally verifying the safety of a simple autonomous car controlled by a neural network, operating in a stochastic environment with imperfect information.

2012 ACM Subject Classification Software and its engineering → Domain specific languages; Software and its engineering → Functional languages; Software and its engineering → Formal software verification; Computing methodologies → Neural networks; Theory of computation → Type theory; Theory of computation → Logic and verification; Computing methodologies → Vagueness and fuzzy logic; Computer systems organization → Embedded and cyber-physical systems

Keywords and phrases Neural Network Verification, Types, Interactive Theorem Provers.

Digital Object Identifier 10.4230/LIPIcs.FSCD.2025.34

Category Invited Talk

Related Version *Preliminary Draft Version*: `https://arxiv.org/abs/2401.06379` [23]

Supplementary Material *Vehicle Language, User Manual and Tutorials (Source Code)*: `https://github.com/vehicle-lang/vehicle`

Funding The authors acknowledge support of the EPSRC grant AISEC: AI Secure and Explainable by Construction (EP/T026960/1, EP/T027037/1, EP/T026960/1). Slusarz acknowledges EPSRC DTA studentship awarded by Heriot-Watt University.

1 Introduction

With the proliferation of neuro-symbolic systems that blend machine learning with symbolic reasoning, the formal verification of the reliability and safety of such systems is an increasingly important concern [57, 67]. Examples include: ensuring the correctness of decision-making software (e.g. insurance assessments [79]) where symbolic software delegate certain cases to trained neural models, and proving the safety of cyber-physical systems (e.g. cars [7] and drones [69]) where neural agents must be proved safe with respect to a symbolic representation of the environment in which they act. Unfortunately, the non-interpretable nature of neural networks means that reasoning formally about these systems is significantly harder than reasoning about purely symbolic systems. Despite this, the formal verification community has achieved notable successes, including the development of automatic theorem provers specialised for reasoning about neural network components in isolation [13], and proving reachability results for neuro-symbolic systems [58]. Nonetheless, current efforts verifying neuro-symbolic systems in general still face substantial challenges, with inconsistencies arising between different stages of training, implementation, verification and deployment [19].

The contributions of this paper are as follows. In Section 2 we propose a general decomposition of the problem of training, constructing and verifying neural-symbolic systems. This decomposition reveals the difficulty of integrating proofs about the neural components with proofs about the symbolic components, which we call *the embedding gap*. In particular, in the general case, we argue that neither interactive theorem provers nor existing automatic theorem provers (even those specialised in verifying neural networks) are suitable for carrying out this integration step. We illustrate the applicability of our analysis by describing a proof of temporal correctness of a simple autonomous car model operating in a non-deterministic, imperfect information environment.

In Section 4, we present our Vehicle tool which is designed to enable the verification of neuro-symbolic systems by facilitating the decomposition we identified in the previous section and to close the embedding gap. The core of Vehicle is a high-level, dependently-typed language designed for writing specifications for the neural components of neuro-symbolic systems. It is optimised for expressivity and readability with support for tensors, neural networks, large datasets, first-class quantifiers and higher-order functions. The Vehicle compiler then translates these specifications to i) machine learning frameworks for training of the neural components, ii) automatic theorem provers for verification of the neural components and iii) interactive theorem provers where proofs about the neural components can be integrated with proofs about the symbolic components. This paper also explains how, although Vehicle's dependent type-system is used directly when writing specifications in a limited fashion, its primary use is internally to translate code between the different backends and provide clear diagnostic error messages to users when their specifications cannot be compiled to a given backend.

2 Analysing the Problem of Neural-Symbolic Verification

2.1 Decomposing the Problem

We will begin by considering an abstract symbolic program $s(\cdot)$, whose completion requires computing an unknown function $\mathcal{H} : \mathcal{P} \to \mathcal{R}$, where $\mathcal{H}$ maps objects in the *problem space* $\mathcal{P}$ to those in the *result space* $\mathcal{R}$. The sets $\mathcal{P}$ and $\mathcal{R}$ may contain a mixture of discrete and continuous components, and crucially are semantically rich, by which we mean they refer to quantities interpretable by humans (e.g. measurements in real world units, images, text etc.).

As $\mathcal{H}$ is a complex function, the goal is to train a neural network to approximate it. However, neural networks can only learn functions between continuous spaces and perform best when the input data is normalised, whereas in general $\mathcal{P}$ and $\mathcal{R}$ may contain a mix of discrete and unnormalised continuous values. The standard approach is to construct an embedding function $e : \mathcal{P} \to \mathbb{R}^m$ and an unembedding function $u : \mathbb{R}^n \to \mathcal{R}$ that map the semantically meaningful objects to and from values in a continuous vector space, and then train a neural network $f : \mathbb{R}^m \to \mathbb{R}^n$ such that $u \circ f \circ e \approx \mathcal{H}$. We will refer to $\mathbb{R}^m$ and $\mathbb{R}^n$ as the *input space* and *output space* respectively. Unlike objects in the problem/result spaces, in general, embedded objects in the input/output spaces are unitless, normalised quantities and are therefore not semantically interpretable. Furthermore, for real systems m and n may be very large, e.g. for image classification networks m will correspond to the number of pixels.

The completed neuro-symbolic program is then modelled as $s(u \circ f \circ e)$. Our end goal is to prove that $s(u \circ f \circ e)$ satisfies some property Ψ, which we will refer to as the *system property*. The natural way to proceed is to establish a *solution property* Φ and a *network property* Ξ such that the proof of Ψ is decomposable into the following three lemmas:

$$\forall h.\ \Phi(h) \Rightarrow \Psi(s(h)) \tag{1}$$

$$\forall g.\ \Xi(g) \Rightarrow \Phi(u \circ g \circ e) \tag{2}$$

$$\Xi(f) \tag{3}$$

i.e. Lemma 1 proves that the system satisifes the property Ψ for *any* implementation of $\mathcal{H}$ that satisfies Φ. Crucially, this property requires only reasoning about the symbolic portion of the system. Lemma 2 links the symbolic and neural components of the proof by proving that if any network satisfies Ξ then when composed with the embedding functions it satisfies Φ. Finally Lemma 3 proves that the actual concrete network f obeys the network property Ξ. Together they can be composed in the obvious way to show that the neuro-symbolic program $s(u \circ f \circ e)$ obeys the program property Ψ.

2.2 The Embedding Gap

We now discuss how we can implement this proof strategy, starting with finding a suitable property Φ and proving Lemma 1 which reasons about the symbolic component of the system. Determining what property Φ should be will usually require deep expertise in the problem domain, but fundamentally requires no new insights or methodology as we can rely on insights from the formal verification community which has many decades of experience in decomposing proofs about symbolic systems down into constituent parts. Likewise, once Φ has been found, the community is well placed to prove results of this form using a variety of powerful interactive theorem provers (ITPs) (e.g. [9, 65]).

We will come back to methods for finding a suitable property Ξ and proving Lemma 2 later. Instead we turn our attention to Lemma 3, the proof about the neural component of the system. Assuming one does have a suitable Ξ, experience has shown that proving properties about neural networks directly in an ITP is challenging. The first issue is that the modular reasoning that ITPs excel at is not well suited to the non-interpretable and semantically non-compositional nature of neural networks. Furthermore the sheer size of the networks, often millions or billions of parameters [44, 80], make even representing the network, let alone proving anything about it, impractical in an ITP. For example, the largest neural network verified to date purely in an ITP is a few hundreds of neurons (or a few thousand of weights) [8, 14, 27, 29]. In contrast, the automated theorem prover (ATP) community has been significantly more successful at proving properties of the form of Lemma 3. Starting

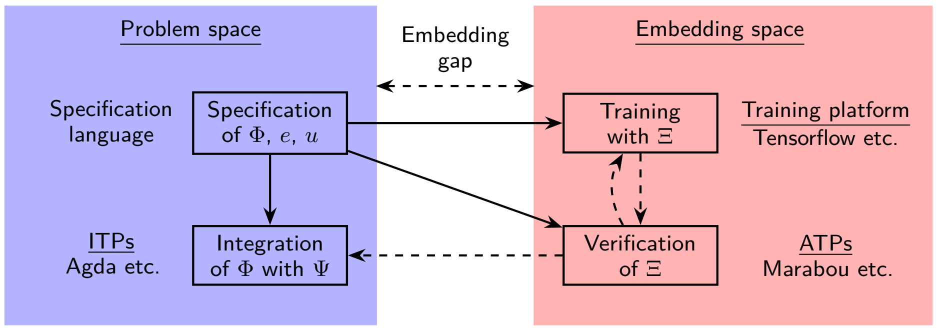

Figure 1 The architecture of VEHICLE for neuro-symbolic program verification. Dashed lines indicate information flow and solid lines automatic compilation.

with Reluplex [51], the community has rapidly developed highly specialised SMT and abstract interpretation-based solvers which are capable of verifying properties of neural networks with up to millions of neurons [52, 63, 78, 81]. There is, however, a further consideration. Unlike conventional software which is usually at least morally (if not actually) correct at verification time, neural networks often struggle to learn property Ξ from data alone [74]. Consequently, we also need property Ξ to influence the training of the network f, e.g. using techniques such as differentiable logic [33] and linear temporal logic-based reward functions [40].

This leaves the problem of how to link the proofs about the symbolic and the neural components of the system, by finding a suitable property Φ and proving Lemma 2. Firstly, even finding a suitable Ξ is difficult as it refers to a semantically uninterpretable input and output spaces. This strongly suggests that we need a method of automatically deriving it from Φ and the embedding functions. Suppose we did have such an automatic procedure. ATPs are ill-suited to proving Lemma 2, as they are not designed to reason about a) the discrete components present in $\mathcal{P}$ and $\mathcal{R}$ and b) the arbitrary computation present in the embedding functions u and e. Unfortunately, ITPs are equally ill-suited to proving Lemma 2 it would require the user to manually write down and then reason about property Φ in the ITP directly (remember Φ is often uninterpretable and scales with the size of the embedding space i.e. potentially tens of thousands of inputs). We call this lack of practical methodology for establishing the results that link the symbolic and neural components of the proof the *embedding gap*.

Given the analysis above, it is clear that in general to construct and verify a neuro-symbolic program, we need machine learning frameworks, ATPs and ITPs to work together. Unfortunately, usually each of these have their own specialised input formats and semantics and currently the default approach is to write out the specification three times for the three different tools. This is deeply suboptimal as it requires an informal judgement that each of the three representations encode the same property.

2.3 Our Vision of the Solution

Figure 1 shows our vision of a tool that overcomes these problems and enables the *general* verification of neuro-symbolic systems. Firstly, users should verify Lemma 1 about the symbolic component of the system using whichever existing ITP best meets their needs. Next the user expresses the specification Φ and the embedding functions e and u in terms of the semantically-meaningful problem space using a suitable domain-specific language (DSL). This specification of Φ is then automatically compiled down to representations of Ξ in the

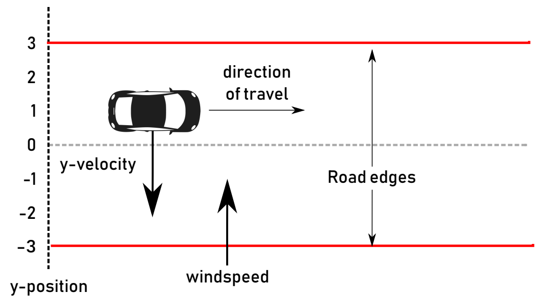

Figure 2 A simple model of an autonomous car compensating for a cross-wind.

semantically uninterpretable input/output spaces suitable for i) training the network in the user's machine learning framework of the choice and ii) verifying the output of training using an appropriate ATP. Once a network f has been trained that the ATPs can prove satisfies $\Xi(f)$, the proof of $\Phi(u \circ f \circ e)$ should be constructed automatically and returned to the ITP.

There are several advantages to such a hypothetical tool. Firstly, the user only has to express the specification once and the tool automatically generates provably equivalent representations suitable for each of the backends. Secondly, the specifications of the neural component Φ are written in terms of the semantically meaningful problem and result spaces $\mathcal{P}$ and $\mathcal{R}$. This means they can be read and checked by experts in the problem domain who may not know anything about machine learning.

3 A Concrete Example

In Section 4 we will introduce our tool Vehicle which implements a large proportion of our vision from Section 2.3. However, before we do so, we will now give an example of a simple, concrete verification problem that illustrates our proposed decomposition described in Section 2 and will assist our explanations of Vehicle's operation in Section 4.

As illustrated in Figure 2, we use a modified version of the verification problem presented by Boyer, Green and Moore [12]. An autonomous car is travelling along a straight road of width 6 parallel to the x-axis, with a varying cross-wind that blows perpendicular to the x-axis. The car has an imperfect sensor that provides noisy measurements of its position on the y-axis, and can change its velocity with respect to the y-axis in response. The car's controller takes in both the current sensor reading and the previous sensor reading and its goal is to keep the car on the road. The desired system safety property that we would like to prove is as follows:

If the wind-speed never shifts by more than 1 per unit time and the sensor is never off by more than 0.25 units then the car will never leave the road.

Note that this control problem involves both stochasticity via the fluctuating wind-speed and imperfect information via the error on the sensor reading.

3.1 Symbolic Component

In order to prove the system property above, it is necessary to first construct a symbolic model of the behaviour of the system (i.e., $s(\cdot)$ in our analysis in Section 2). We discretise the model as in [12], and then formalise it in Agda, an interactive theorem prover. As discussed in Section 2.3, neither the discretisation nor the use of Agda are relevant to the central proposal of this paper. We could equally have chosen to create a continuous model of the system based on differential equations in alternative systems such as Rocq or KeYmaera X.

The state of the system consists of the current wind speed, the position and velocity of the car and the most recent sensor reading. An oracle provides updates in the form of observations consisting of the shift in wind speed and the error on the sensor reading. The third component is a controller that takes as input the current and previous sensor readings and produces a recommended change in velocity:

```
record State : Set where
  constructor state
  field
    windSpeed : ℚ
    position  : ℚ
    velocity  : ℚ
    sensor    : ℚ
```

```
record Observation : Set where
  constructor observe
  field
    windShift   : ℚ
    sensorError : ℚ

controller : ℚ → ℚ → ℚ
```

Using these components, we can define the evolution of the system as:

```
nextState : Observation → State → State
nextState o s = state newWindSpeed newPosition newVelocity newSensor
  where
  newWindSpeed = windSpeed s + windShift o
  newPosition  = position s + velocity s + newWindSpeed
  newSensor    = newPosition + sensorError o
  newVelocity  = velocity s + controller newSensor (sensor s)

finalState : List Observation → State
finalState xs = foldr nextState initialState xs
```

Given suitable encodings of ValidObservation and OnRoad, the system safety property (i.e., Ψ in our analysis in Section 2) can be formalised as follows:

```
finalState-onRoad : ∀ xs → All ValidObservation xs → OnRoad (finalState xs)
```

This statement can be proved in Agda by induction over the list of observations, and can be found in the supplementary material. The proof crucially requires the controller to satisfy the following property (Φ in our analysis in Section 2):

```
controller-lemma : ∀ x y → ∣ x ∣ ≤ 3.25 → ∣ y ∣ ≤ 3.25 → ∣ controller x y + 2 * x - y ∣ < 1.25
```

This says that if both the current and previous sensor readings say that the car is less 3.25 metres from the centre of the road, then the sum of the output of the controller and twice the current sensor reading minus the previous sensor reading must be less than 1.25. The goal is to implement the function controller with a neural network that provably satisfies controller-lemma.

3.2 Neural Component

As this is a simple control problem, we choose 3 densely connected layers as the architecture for our neural network controller. In terms of the generic problem decomposition we described in Section 2, the embedding function e normalises the semantically meaningful problem space inputs measured in metres in the range $[-4, 4]$ to the range $[0, 1]$ in the embedding space.

```
type Input = Tensor Rat [2]
currentPosition = 0
previousSensor = 1

type Output = Tensor Rat [1]
velocity = 0

@network
controller : Input -> Output

normalise : Input -> Input
normalise x = forall i . (x ! i + 4.0) / 8.0

safeInput : Input -> Bool
safeInput x = -3.25 <= x ! currentSensor <= 3.25 and
              -3.25 <= x ! previousSensor <= 3.25

safeOutput : Output -> Bool
safeOutput x = let y = controller (normalise x) ! velocity in
    -1.25 < y + 2 * x ! currentSensor - x ! previousSensor < 1.25

@property
safe : Bool
safe = forall x . safeInput x => safeOutput x
```

Figure 3 The safety property for the car's neural network controller expressed in Vehicle surface syntax .

The unembedding function u is the identity function. Again, as discussed in Section 2.3, neither the choice of architecture nor the choice of embedding functions are relevant to the central proposal in the paper.

4 The Vehicle Tool

Our Vehicle tool implements the vision we described in Section 2.3: users are provided a simple domain-specific language with types to write a single specification about the neural component of the neuro-symbolic system (see Figure 3). Vehicle then compiles this specification into forms suitable for both training and verification and can export the verified specification to ITPs. Vehicle can be installed by running "`pip install vehiclepy`", which provides both a Python and a command line interface. A user manual and tutorials can be found online [20, 24].

4.1 Specification Language

The Vehicle Condition Language (VCL) is designed for writing high-level, problem space specifications for neural networks. At its core is a dependently-typed λ-calculus extended with operations for logic, arithmetic and manipulating tensors. The abstraction capabilities of the λ-calculus enable users to write modular and reusable specifications, while the dependent

$$
\begin{array}{l}
\langle \mathit{prog} \rangle \ni p \ ::= [\,] \mid \mathrm{d} :: p \\
\\
\langle \mathit{decl} \rangle \ni d \ ::= \\
\quad \mid \ \texttt{function}\ \langle \mathit{id} \rangle : e = e \\
\quad \mid \ \texttt{@network}\ \langle \mathit{id} \rangle : e \\
\quad \mid \ \texttt{@dataset}\ \langle \mathit{id} \rangle : e \\
\quad \mid \ \texttt{@parameter}\ \langle \mathit{id} \rangle : e \\
\quad \mid \ \texttt{@property}\ \langle \mathit{id} \rangle : e = e
\end{array}
\qquad
\begin{array}{l}
\langle \mathit{expr} \rangle \ni e \ ::= \\
\quad \texttt{Type} \mid \Pi(x : e)\,.\,e \mid \lambda\,(x : e)\,.\,e \mid x \mid e\ e \mid \\
\quad \texttt{Bool} \mid \texttt{true} \mid \texttt{false} \mid \texttt{not}\ e \mid e\ \texttt{and}\ e \mid e\ \texttt{or}\ e \mid \\
\quad \texttt{forall}\ (x : e)\,.\ e \mid \texttt{exists}\ (x : e)\,.\ e \mid \\
\quad \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \\
\quad \texttt{Real} \mid r \in \mathbb{R} \mid e \texttt{+} e \mid e \texttt{*} e \mid e \texttt{==} e \mid e \texttt{!=} e \mid e \texttt{<=} e \mid e \texttt{<} e \mid \\
\quad \texttt{Tensor}\ e\ e \mid \texttt{Index}\ e \mid n \in \mathbb{N} \mid [e, \ldots, e] \mid e\ \texttt{!}\ e \mid \\
\quad \texttt{foreach}\ (x : \mathrm{n})\ .\ e \mid \texttt{foldr}\ e\ e\ e
\end{array}
$$

Figure 4 Syntax for the core calculus of Vehicle

types allow tensor dimensions to be tracked at the type level, preventing common specification errors such as dimensions mismatches and out-of-bounds indexing. A standard instance resolution mechanism allows for the overloading of operators in the surface syntax. The syntax of the core language is shown in Figure 4.

A Vehicle program consists of a list of declarations. There are four non-standard declaration types in VCL that describe how a specification links to the outside world. Firstly, `@network` declarations introduce an abstract neural network into scope. Next `@parameter` and `@dataset` declarations can be used to introduce external values that either may not be known in advance or are too big to represent directly in the specification. Crucially, only the user only needs to provide the types of these declarations when writing the specification. As discussed in Section 4.2, the user provdies the compiler with their actual implementation or value at compile time. Finally, `@property` declarations are used to explicitly designate a constraint the neural networks are expected to satisfy. A complete description of the syntax is available in the user manual [24].

Figure 3 illustrates how the specification, Φ, for the car controller from Section 3 is expressed in Vehicle. The embedding function e is represented by the "normalise" function on Lines 11-12. Crucially, the safety conditions specified on Lines 14-20 are written in terms of the problem space, using units such as metres, rather than being expressed in the embedding space. This makes them interpretable and meaningful to readers of the specification.

4.2 The Vehicle Compiler

The Vehicle compiler translates VCL specifications into formats suitable for three different backends: Tensorflow loss functions for training, Marabou queries for verification and Agda code for integration with the proof about the surrounding symbolic system. The compiler is implemented in Haskell, and its overall architecture is illustrated in Figure 5.

4.2.1 The Verification Backend

Given a trained network f and a Vehicle specification representing Φ, u and e, the purpose of the verifier backend is to determine if $\Phi(u \circ f \circ e)$ holds and if not to find a counter-example. Vehicle accomplishes this goal by compiling the specification into a set of satisfiability queries for the Marabou verifier [52].

Figure 6 shows the two queries generated from the specification in Figure 3. The queries are equisatifiable to the original specification in the sense that a neural network satisfies the specification if and only if no satisfying assignment to variables x_0 and x_1 can be found for either query. Collectively, the queries represent the property Ξ. It is important to note that, unlike the original specification which referenced physical quantities such as speed and

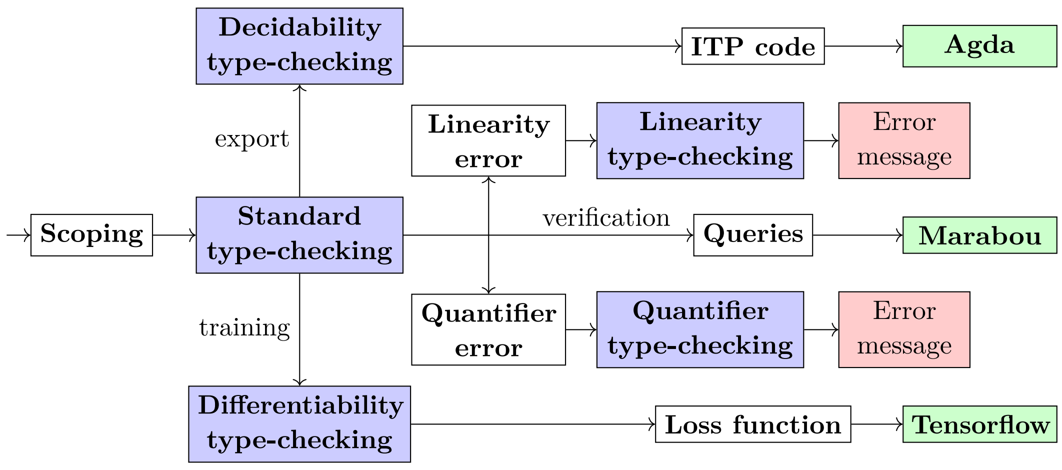

Figure 5 Overview of the VEHICLE compiler. The same unification-based type-checker is re-used in five different ways to provide support for the different backends.

```
x0 >= 0.09375
x0 <= 0.90625
x1 >= 0.09375
x1 <= 0.90625
16.0x0 - 8.0x1 + y0 <= 2.75
```

```
x0 >= 0.09375
x0 <= 0.90625
x1 >= 0.09375
x1 <= 0.90625
-16.0x0 + 8.0x1 - y0 <= -5.25
```

Figure 6 The two Marabou queries, representing Ξ, generated by the VEHICLE compiler from the specification in Figure 3. Both queries are implicitly existentially quantified over the variables x_0 and x_1 which represent the inputs to the neural network, and y_0 which represents the output of the network. These queries are not seen by a VEHICLE user in normal operation.

distances, the Marabou queries refer solely to quantities in the input/output spaces. As a result, the variables and numeric values in these queries are not directly interpretable in terms of the symbolic model of the environment.

The compilation algorithm interleaves a normalisation-by-evaluation algorithm [10] with a procedure for the elimination of the quantified problem-space variables. These variables are eliminated by rewriting them in terms of the variables representing the input and outputs of the networks using Gaussian [6] and Fourier-Motzkin [25] elimination, producing a final representation entirely in the network's input/output space. Notably, the algorithm supports complex specifications , including those with nested applications of multiple networks and quantifiers embedded within the conditions of ‘if‘ statements. Ensuring that this procedure remains computationally efficient, particularly for networks with tens of thousands of inputs and outputs, introduces significant complexity. A full description of the compilation procedure is provided in [22].

The output of the compilation process is a tree structure, where each internal node represents a conjunction or disjunction, and the leaves correspond to individual verifier queries. The tree is then stored on disk as the basis of the “proof cache". When verification is requested, VEHICLE loads the cache and traverses the tree, invoking Marabou on each query to decide if there exists a satisfying assignment for the current network and query. The outcome of each query is recorded in the cache. If any counterexamples for Ξ are found in the input-output space, the compiler lifts them to counterexamples for Φ in the origin problem/result space. Depending on the output of Marabou for each query, Vehicle can then determine that the statement $\Phi(u \circ f \circ e)$ holds.

It is important to emphasise that VEHICLE does not expand the class of verifiable specifications beyond what is supported by Marabou. Instead, it enhances the interpretability of those specifications and streamlines the process of generating verification queries. Critically, Marabou only supports linear constraints with existential quantifiers. Therefore if a user writes a VEHICLE specification involving non-linear constraints or alternating quantifiers VEHICLE will produce an error.

In such cases, explaining why a specification is not verifiable is an important usability feature. Some of example specifications exceed 300 lines, making it difficult to identify where non-linearities or alternating quantifiers have been introduced. To provide meaningful error messages, we leverage VEHICLE's dependent type-checker and instance resolution mechanism to construct a proof of why the original specification is non-linear or has alternating quantifiers. In particular, we replace all types with meta-variables and then re-type check the specification according to a new set of typing rules for the builtin operations. Because type-checking is constructive, the type-checker produces a proof term that the compiler can use to generate clear, actionable explanations. Full details of this procedure are available in [21].

4.2.2 The ITP Backend

The ITP backend is responsible for exporting the the proof of $\Phi(u \circ f \circ e)$ to an ITP, enabling it to be combined with the proof of Lemma 1 in order to establish $\Psi(s(u \circ f \circ e))$ in the ITP. Currently, VEHICLE supports exporting specifications to Agda, with Rocq support nearing completion. The Agda code generated for the specification in Figure 3 is shown in Figure 7. Given that most ITP languages are more expressive than VEHICLE this translation may appear straightforward, however there are two main challenges.

Firstly, as discussed in Section 2, in the general case the network and the proof of correctness cannot be represented directly in Agda itself for performance reasons. Instead, as shown in Figure 7, the network is included as a postulate and an Agda macro is used to delegate checking the proof to Vehicle. The new challenge lies in preserving Agda's interactivity while ensuring the integrity of the proof. In particular, the Agda proof should fail if the ONNX network file on disk is changed (for example, if the network is retrained after verification). At the same time, Agda's often invoke Agda's type-checker several times per minute and re-verifying the proof on every interaction is impractical. The solution is the proof cache discussed in Section 4.2.1. Alongside the query tree, VEHICLE stores the hash and location of all networks and datasets used during verifier query compilation within the proof cache. When Agda queries the validity of the proof, VEHICLE uses these hashes to verify the integrity of the verification result without having to invoke Marabou again. As we discuss further in Section 5, we are exploring the feasability of replacing these hashes with efficiently checkable proof-certificates in the style of [28].

The second challenge arises because users write specifications as Boolean expressions in VEHICLE, implicitly assuming that the property is decidable. However, ITPs such as Agda require the specification to be encoded at the type-level (e.g. safe in Figure 7 and finalState-onRoad in Section 3), where decidability does not generally hold. Despite this, the conditional term in an 'If' expression must be decidable and the therefore the same term in the VEHICLE specification can be used at both the type level and the Boolean level in the generated ITP code. To determine how to translate the term, we use the same approach as generating error messages for specificationc containing non-linear or alternating quantifiers described in Section 4.2.1. Namely, we leverage VEHICLE's dependently-type checker and

```
module ControllerSpecification where

InputVector : Set                                  OutputVector : Set
InputVector = Tensor ℚ (2 :: [])                   OutputVector = Tensor ℚ (1 :: [])

currentSensor : Fin 2                              velocity : Fin 1
currentSensor = # 0                                velocity = # 0

previousSensor : Fin 2
previousSensor = # 1

postulate controller : InputVector → OutputVector

normalise : InputVector → InputVector
normalise x = λ i → (x i ℚ.+ 4) ℚ.÷ 8

SafeInput : InputVector → Set
SafeInput x   = (ℚ.- 3.25 ℚ.≤ currentSensor x × currentSensor x ℚ.≤ 3.25)
              × (ℚ.- 3.25 ℚ.≤ previousSensor x × previousSensor x ℚ.≤ 3.25)

SafeOutput : InputVector → Set
SafeOutput x =
  let y = controller (normalise x) velocity in
  ℚ.- 1.25 ℚ.< (y ℚ.+ 2 ℚ.* currentSensor x) ℚ.- previousSensor x
  × (y ℚ.+ 2 ℚ.* currentSensor x) ℚ.- previousSensor x ℚ.< 1.25

abstract
  safe : ∀ x → SafeInput x → SafeOutput x
  safe = checkVehicleProperty record
    { proofCache = "path/to/property/file.vclp"
    }
```

Figure 7 Agda code by VEHICLE when exporting the specification in Figure 3. Import statements are omitted.

instance resolution, replacing all Boolean types with meta-variables and then re-solving according to new typing rules for the Boolean builtin operations.

4.2.3 The Training Backend

The final backend enables users to train a network to satisfy a VEHICLE specification. Numerous techniques have been proposed for incorporating specifications into the training process [26, 39]. VEHICLE implements a method known as *differentiable logic* (DL) [33, 55, 72], which converts a Boolean-valued specification into a loss function that penalises deviations from the desired property. The resulting loss function is differentiable almost everywhere with respect to the network weights, allowing it to be used with standard gradient descent algorithms during training. For a broader discussion on the machine-learning justification

```
1  import vehicle_lang as vcl
2
3  loss_fn = vcl.load_loss_function(
4      specification_path="controller-spec.vcl",
5      property_name="safe",
6      target=vcl.DifferentiableLogic.DL2,
7  )
8
9  # Standard code to create and train neural network
10 model = ...
11 for epoch in range(num_epochs):
12     with tf.GradientTape() as tape:
13         loss = loss_fn(controller=model)
14     grads = tape.gradient(loss, model.trainable_weights)
15     optimizer.apply_gradients(zip(grads, model.trainable_weights))
```

(a) Python code for using the specification in Figure 3 to train a model. Lines 3-7 contain the Vehicle-specific code needed to generate the loss function. Lines 10-15 are standard code to train a neural network given an arbitrary loss function.

```
def lossFn(controller):
  return sample(
    samples=10,
    domain=[[0.09375,0.09375],[0.90625,0.90625]],
    body=
      lambda v:
        sum(0.0,
          (sum(0.0, -2.75 - controller([xi + 4.0) / 8.0 for xi in v])[0]
      + 2.0 * v[0] - v[1])) +
            sum(0.0, - 2.75 + controller([xi + 4.0) / 8.0 for xi in v])[0]
       + 2.0 * v[0] - v[1]))))
  )
```

(b) The implementation of the loss function generated automatically by VEHICLE in Figure 8a. This code is never normally seen by a VEHICLE user.

Figure 8 VEHICLE's training backend.

for using differentiable logic to generate optimisation objectives, see [34].

Differentiable logic was chosen for two key reasons: a) its generality - depending on the differentiable logic used, any well-typed VEHICLE specification can be converted to a corresponding loss function; and b) its flexibility - the specification-based training can either be integrated into standard data-driven or reinforcement-based learning workflows or can be applied as an additional fine-tuning step post-training.

Concretely, given a `@property p`, VEHICLE compiles `p` into a pure function that takes external resources (i.e. networks, datasets and parameters) as inputs and returns a numeric output representing "how false" the property `p` is. The exact translation method depends on the chosen differential logic. Numerous logics have been proposed, and VEHICLE currently implements DL2 [33], Gödel, Product, Łukasiewicz and Yager logics [55].

The compilation process procedes in two steps. Firstly, the specification must be split into two parts: the constraints on the quantified variables and the constraints on the network's behaviour. Critically, the latter needs to be translated into a real-valued formula using the differentiable logic. To do this, we again use the type-checker and instance resolution in a similar fashion to that described in Sections 4.2.1 & 4.2.2. Next, the same standard normalisation-by-evaluation algorithm as used by the verification algorithm is interleaved with a procedure for splitting the differentiable and the non-differentiable constraints. The former are translated to numeric operations according to the differentiable logic being used. The latter represents the domain of the quantified variables (i.e. the set of values they are allowed to assume) and therefore are attached to the quantifier node in the AST to allow for efficient sampling as described in [72].

Figure 8a shows the Python code required to be written by the user to train the network using the specification from Section 3, while Figure 8b shows the code that is generated behind the scenes to implement the loss function. Note that although training via differentiable logic is a very general technique and has shown to be effective in practice [34, 35], it does not guarantee that the network will satisfy the specification after training. How to do so is still an open problem.

4.2.4 Soundness

Given the complexity of the Vehicle system and its diverse backends, it is important to ensure its overall soundness. To this end, we have developed a formal semantics for the Vehicle core language, as well as for the target languages used by both the training and the verifier backends. Based on this foundation, we have proved not only the soundness of the compilation to the two backends, but also that the loss function and the verifier queries generated are logically equivalent in some general sense. This ensures that the propery being trained for is the same as that which we are verifying. These proofs have been formalised in Agda, and can be found at [5]. We have not formalised the correctness of the ITP backend.

5 Related and Future Work

Given the wide range of areas that Vehicle intersects with – including formal verification, program synthesis, neural network training, and theorem proving – there is a substantial body of related work. In this section, we highlight some of the most relevant contributions.

Programming Language Interfaces for Neural Network Verification. The need for more conceptual and robust tools and programming language practices has recently been flagged as one of the biggest challenges in enabling the future development of neural network verification [19]. Apart from the embedding gap, four problems have been flagged as substantial in [19]: the lack of rigorous semantics of specification languages deployed in neural network verification, formalisation and generation of proof certificates, the implementation gap, and support for property-driven training. Vehicle hopes to contribute to resolving most of them. Other frameworks that provide neural network specification DSLs similar to that of Vehicle include DNNV [70] and CAISAR [38]. Unlike Vehicle, much of their focus is on improving interoperability between different ATPs for neural network verification, and they do not solve the problem of the embedding gap or integrate training.

Explainability and Specifications in Machine Learning. Vehicle's methodology applies in domains where a suitable specification is available. Therefore it is not currently easily applicable to domains such as NLP and Computer Vision which defy clearly defined correctness criteria. See [17, 16] for more detailed discussion of this issue. Vehicle is therefore complementary to work such as [41] and [66] which obtain formal statistical guarantees about neural networks used as sensors in cyber-physical systems.

Cyber-Physical System Verification. Cyber-Physical Systems (CPS) with machine learning components is an important safety-critical use case for neural network verification. As shown in our running example, a neural network may be utilized as a feedback controller for some plant model, typically represented as ordinary differential equations (ODEs) or generalizations thereof like hybrid automata. These are known as *neural networks control systems (NNCS).* The annual International Competition on Verifying Continuous and Hybrid Systems (ARCH-COMP) held with the Applied Verification for Continuous and Hybrid Systems (ARCH) workshop has a category for this problem class, known as the AI and NNCS (AINNCS) category [61, 50, 49, 56, 59]. Several approaches for addressing the NNCS verification problem have been developed, such as implemented within software tools like CORA [53], JuliaReach [11], NNV [77, 60], OVERT [71], POLAR [42], Sherlock [31, 30], ReachNN* [43, 32], VenMAS [3], and Verisig [47, 46, 45]. More broadly, researchers have considered several strategies for the specification of properties of CPS with neural network components [36, 4, 15]. These cover significant challenges in the CPS domain, ranging from classical software verification problems to real-time systems concerns, scalability, as well as finding suitable specifications [68, 76, 75, 62]. Crucially however, many of these techniques can not be easily linked to more general purpose tools such as ITPs for reasoning about the environments. We believe this area would also benefit from a more principled programming language support, and languages like VEHICLE can provide a trustworthy infrastructure for consistent specifications of these complex systems.

Neuro-Symbolic Programs and Proof-Carrying Code. Finally, our work on VEHICLE also relates to the nascent field of *neuro-symbolic programming*, seen as a collection of methods of merging machine learning code and standard (symbolic) code [18]. In particular, VEHICLE can be seen as a step towards the goal of enabling a *proof-carrying neuro-symbolic code* [54]. The idea is to use a combination of formal methods and compilation techniques to enable light-weight verification of complex neuro-symbolic interfaces in the style of *self-certifying code* [64].

Our priorities for improving Vehicle include:

- **Proof Certificates**. Currently the ITP must trust Vehicle's assertion that the network satisfies the specification. As discussed in Section 2.2, directly representing large neural networks within the ITP is likely infeasible. However, we are exploring ways to adapt Vehicle and the ATPs to generate proof certificates that can be efficiently checked by the ITP itself. The feasibility of checking ATP certificates was demonstrated in [28].
- **ITP Backends**. VEHICLE was designed with a view of providing a principled (and sound) way of interfacing to many ITPs, depending on the demands of the 'symbolic' component verification. For example, industrial ITPs such as Imandra offer stronger automation and libraries that support infinite-precision reals as well as floats; Rocq has extensive Measure theory libraries [2, 1]; KeYmaera X [37] is designed for reasoning about cyber-physical systems with continuous dynamics. Such features may facilitate future CPS verification projects, where VEHICLE can help to interface with neural network verifiers. Recent Rocq backend for Vehicle has proven easy to implement [73], and has demonstrated VEHICLE's readiness for future ITP extensions.
- **Numeric Quantisation**. Currently the VEHICLE syntax and semantics assume the neural networks operate over real numbers. However, in practice neural networks are implemented using quantised floating point values, with a precision of anywhere between 4 and 32 bits. This mismatch in the semantics has been shown to affect the soundness of

the neural network verifiers themselves [48, 19]. How best to address the quantisation issue during verification is an open problem.

6 Conclusions

In this paper we have identified the embedding gap as an existing problem in the verification of neural-symbolic programs and described VEHICLE, the first tool that aims to bridge that gap. We have shown how VEHICLE facilitates proofs about the correctness of neuro-symbolic programs by linking specifications to training frameworks, verifiers and ITPs. We have also demonstrated its utility by verifying the correctness of a neuro-symbolic car controller. We believe this to be the first ever modular proof of the complete verification of a neuro-symbolic program that utilises both ATPs and ITPs.

Our example is, of course, a toy scenario that was primarily chosen because it is small enough to fit in this paper. In a real-world scenario, the environmental dynamics are far more complicated and the car controller will have other objectives such as reaching way points and obstacle avoidance. Therefore we believe one of the overall challenges in this field is to work out how to construct our neuro-symbolic systems so that the safety critical properties (i.e. staying on the road, collision avoidance) are formally verifiable, while allowing the neural components to optimise for the non-safety-critical goals.

7 Contribution statement

Conceptualisation and analysis by Daggitt, Kokke, Atkey and Komendantskaya with help from Arnaboldi. Implementation of VEHICLE by Daggitt and Kokke with help from Atkey and Slursarz. Manuscript preparation by Daggitt, Komendantskaya and Atkey.

References

1 Affeldt, R., Bertot, Y., Cohen, C., Kerjean, M., Mahboubi, A., Rouhling, D., Roux, P., Sakaguchi, K., Stone, Z., Strub, P.Y., et al.: Mathcomp-analysis: Mathematical components compliant analysis library (2022)

2 Affeldt, R., Cohen, C.: Measure construction by extension in dependent type theory with application to integration. J. Autom. Reason. **67**(3), 28 (2023). https://doi.org/10.1007/S10817-023-09671-5, `https://doi.org/10.1007/s10817-023-09671-5`

3 Akintunde, M.E., Botoeva, E., Kouvaros, P., Lomuscio, A.: Formal verification of neural agents in non-deterministic environments. In: International Conference on Autonomous Agents and Multiagent Systems, AAMAS. pp. 25–33 (2020)

4 Astorga, A., Hsieh, C., Madhusudan, P., Mitra, S.: Perception contracts for safety of ml-enabled systems. Proc. ACM Program. Lang. **7**(OOPSLA2) (Oct 2023). https://doi.org/10.1145/3622875

5 Atkey, R., Daggitt, M.L., Kokke, W.: Vehicle formalisation. `https://github.com/vehicle-lang/vehicle-formalisation` (2024)

6 Atkinson, K.: An introduction to numerical analysis. John wiley & sons (1991)

7 Badue, C., Guidolini, R., Carneiro, R.V., Azevedo, P., Cardoso, V.B., Forechi, A., Jesus, L., Berriel, R., Paixao, T.M., Mutz, F., et al.: Self-driving cars: A survey. Expert Systems with Applications **165**, 113816 (2021)

8 Bagnall, A., Stewart, G.: Certifying the True Error: Machine Learning in Coq with Verified Generalization Guarantees. In: Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence. AAAI'19, AAAI Press (2019). https://doi.org/10.1609/aaai.v33i01.33012662

9 Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J.C., Gimenez, E., Herbelin, H., Huet, G., Munoz, C., Murthy, C., et al.: The Coq proof assistant reference manual: Version 6.1. Ph.D. thesis, Inria (1997)

10 Berger, U., Schwichtenberg, H.: An inverse of the evaluation functional for typed lambda-calculus (1991)

11 Bogomolov, S., Forets, M., Frehse, G., Potomkin, K., Schilling, C.: JuliaReach: A toolbox for set-based reachability. In: Proc. of the 22nd ACM International Conference on Hybrid Systems: Computation and Control. p. 39–44 (2019). https://doi.org/10.1145/3302504.3311804

12 Boyer, R.S., Green, M.W., Moore, J.S.: The use of a formal simulator to verify a simple real time control program. In: Beauty Is Our Business, pp. 54–66. Springer (1990)

13 Brix, C., Bak, S., Johnson, T.T., Wu, H.: The fifth international verification of neural networks competition (VNN-COMP 2024): Summary and results. CoRR **abs/2412.19985** (2024). https://doi.org/10.48550/ARXIV.2412.19985, https://doi.org/10.48550/arXiv.2412.19985

14 Brucker, A.D., Stell, A.: Verifying feedforward neural networks for classification in Isabelle/HOL. In: Chechik, M., Katoen, J., Leucker, M. (eds.) Formal Methods - 25th International Symposium, FM 2023, Lübeck, Germany, March 6-10, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14000, pp. 427–444. Springer (2023). https://doi.org/10.1007/978-3-031-27481-7_24

15 Calinescu, R., Imrie, C., Mangal, R., Rodrigues, G.N., Păsăreanu, C., Santana, M.A., Vázquez, G.: Controller synthesis for autonomous systems with deep-learning perception components. IEEE Transactions on Software Engineering **50**(6), 1374–1395 (2024). https://doi.org/10.1109/TSE.2024.3385378

16 Casadio, M., Dinkar, T., Komendantskaya, E., Arnaboldi, L., Daggitt, M.L., Isac, O., Katz, G., Rieser, V., Lemon, O.: Nlp verification: towards a general methodology for certifying robustness. European Journal of Applied Mathematics p. 1–58 (2025). https://doi.org/10.1017/S0956792525000099

17 Casadio, M., Komendantskaya, E., Daggitt, M.L., Kokke, W., Katz, G., Amir, G., Refaeli, I.: Neural network robustness as a verification property: A principled case study. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13371, pp. 219–231. Springer (2022). https://doi.org/10.1007/978-3-031-13185-1_11, https://doi.org/10.1007/978-3-031-13185-1_11

18 Chaudhuri, S., Ellis, K., Polozov, O., Singh, R., Solar-Lezama, A., Yue, Y.: Neurosymbolic programming. Foundations and Trends® in Programming Languages **7**(3), 158–243 (2021). https://doi.org/10.1561/2500000049, http://dx.doi.org/10.1561/2500000049

19 Cordeiro, L.C., Daggitt, M.L., Girard-Satabin, J., Isac, O., Johnson, T.T., Katz, G., Komendantskaya, E., Lemesle, A., Manino, E., Šinkarovs, A., Wu, H.: Neural network verification is a programming language challenge. In: Vafeiadis, V. (ed.) Programming Languages and Systems. pp. 206–235. Springer Nature Switzerland, Cham (2025)

20 Daggitt, M., Kokke, W., Komendantskaya, E., Atkey, R., Arnaboldi, L., Slusarz, N., Casadio, M., Coke, B., Lee, J.: The vehicle tutorial: Neural network verification with vehicle. In: Narodytska, N., Amir, G., Katz, G., Isac, O. (eds.) Proceedings of the 6th Workshop on Formal Methods for ML-Enabled Autonomous Systems. Kalpa Publications in Computing, vol. 16, pp. 1–5. EasyChair (2023). https://doi.org/10.29007/5s2x, https://easychair.org/publications/paper/Rkrv

21 Daggitt, M.L., Atkey, R., Kokke, W., Komendantskaya, E., Arnaboldi, L.: Compiling higher-order specifications to SMT solvers: How to deal with rejection constructively. In: Krebbers, R., Traytel, D., Pientka, B., Zdancewic, S. (eds.) Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023. pp. 102–120. ACM (2023). https://doi.org/10.1145/3573105.3575674, https://doi.org/10.1145/3573105.3575674

22 Daggitt, M.L., Kokke, W., Atkey, R.: Efficient compilation of expressive problem space specifications to neural network solvers. CoRR **abs/2402.01353** (2024). https://doi.org/10.48550/ARXIV.2402.01353, `https://doi.org/10.48550/arXiv.2402.01353`

23 Daggitt, M.L., Kokke, W., Atkey, R., Slusarz, N., Arnaboldi, L., Komendantskaya, E.: Vehicle: Bridging the embedding gap in the verification of neuro-symbolic programs. CoRR **abs/2401.06379** (2024). https://doi.org/10.48550/ARXIV.2401.06379, `https://doi.org/10.48550/arXiv.2401.06379`

24 Daggitt, M.L., Kokke, W., Bob, A., Ślusarz, N., Casadio, M.: Vehicle (2023), `https://github.com/vehicle-lang/vehicle`, accessed on 12.01.2024

25 Dantzig, G.B., Eaves, B.C., et al.: Fourier-Motzkin elimination and its dual. J. Comb. Theory, Ser. A **14**(3), 288–297 (1973)

26 Dash, T., Chitlangia, S., Ahuja, A., Srinivasan, A.: A review of some techniques for inclusion of domain-knowledge into deep neural networks. Scientific Reports **12**(1), 1040 (2022)

27 De Maria, E., Bahrami, A., l'Yvonnet, T., Felty, A., Gaffé, D., Ressouche, A., Grammont, F.: On the use of formal methods to model and verify neuronal archetypes. Frontiers of Computer Science **16**(3), 1–22 (2022)

28 Desmartin, R., Isac, O., Komendantskaya, E., Stark, K., Passmore, G.O., Katz, G.: A certified proof checker for deep neural network verification in Imandra. CoRR **abs/2405.10611** (2024). https://doi.org/10.48550/ARXIV.2405.10611, `https://doi.org/10.48550/arXiv.2405.10611`

29 Desmartin, R., Passmore, G.O., Komendantskaya, E., Daggit, M.: CheckINN: Wide Range Neural Network Verification in Imandra. In: PPDP 2022: 24th International Symposium on Principles and Practice of Declarative Programming, Tbilisi, Georgia, September 20 - 22, 2022. pp. 3:1–3:14. ACM (2022). https://doi.org/10.1145/3551357.3551372, `https://doi.org/10.1145/3551357.3551372`

30 Dutta, S., Chen, X., Sankaranarayanan, S.: Reachability analysis for neural feedback systems using regressive polynomial rule inference. In: ACM International Conference on Hybrid Systems: Computation and Control, HSCC. pp. 157–168 (2019). https://doi.org/10.1145/3302504.3311807

31 Dutta, S., Jha, S., Sankaranarayanan, S., Tiwari, A.: Learning and verification of feedback control systems using feedforward neural networks. IFAC-PapersOnLine **51**(16), 151 – 156 (2018). https://doi.org/10.1016/j.ifacol.2018.08.026, iFAC Conference on Analysis and Design of Hybrid Systems ADHS 2018

32 Fan, J., Huang, C., Li, W., Chen, X., Zhu, Q.: Reachnn*: A tool for reachability analysis ofneural-network controlled systems. In: International Symposium on Automated Technology for Verification and Analysis (ATVA) (2020)

33 Fischer, M., Balunovic, M., Drachsler-Cohen, D., Gehr, T., Zhang, C., Vechev, M.T.: DL2: training and querying neural networks with logic. In: Chaudhuri, K., Salakhutdinov, R. (eds.) Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA. Proceedings of Machine Learning Research, vol. 97, pp. 1931–1941. PMLR (2019), `http://proceedings.mlr.press/v97/fischer19a.html`

34 Flinkow, T., Casadio, M., Kessler, C., Monahan, R., Komendantskaya, E.: A generalised framework for property-driven machine learning (2025), `https://arxiv.org/abs/2505.00466`

35 Flinkow, T., Pearlmutter, B.A., Monahan, R.: Comparing differentiable logics for learning with logical constraints. Science of Computer Programming **244**, 103280 (Sep 2025). https://doi.org/10.1016/j.scico.2025.103280

36 Fremont, D.J., Dreossi, T., Ghosh, S., Yue, X., Sangiovanni-Vincentelli, A.L., Seshia, S.A.: Scenic: a language for scenario specification and scene generation. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 63–78. PLDI 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3314221.3314633

37 Fulton, N., Mitsch, S., Quesel, J.D., Völp, M., Platzer, A.: KeYmaeraX]: An axiomatic tactical theorem prover for hybrid systems. In: Felty, A.P., Middeldorp, A. (eds.) CADE. LNCS, vol. 9195, pp. 527–538. Springer (2015). https://doi.org/10.1007/978-3-319-21401-6_36

38 Girard-Satabin, J., Alberti, M., Bobot, F., Chihani, Z., Lemesle, A.: Caisar: A platform for characterizing artificial intelligence safety and robustness. In: AISafety. CEUR-Workshop Proceedings, Vienne, Austria (Jul 2022), `https://hal.archives-ouvertes.fr/hal-03687211`

39 Giunchiglia, E., Stoian, M.C., Lukasiewicz, T.: Deep Learning with Logical Constraints. In: Raedt, L.D. (ed.) Proceedings of the International Joint Conference on Artificial Intelligence 2022. pp. 5478–5485. ijcai.org (2022). https://doi.org/10.24963/ijcai.2022/767

40 Hasanbeig, H., Kroening, D., Abate, A.: Certified reinforcement learning with logic guidance. Artificial Intelligence **322**, 103949 (2023)

41 Hsieh, C., Li, Y., Sun, D., Joshi, K., Misailovic, S., Mitra, S.: Verifying controllers with vision-based perception using safe approximate abstractions. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **41**(11), 4205–4216 (2022)

42 Huang, C., Fan, J., Chen, X., Li, W., Zhu, Q.: POLAR: A polynomial arithmetic framework for verifying neural-network controlled systems. In: International Symposium on Automated Technology for Verification and Analysis (ATVA) (2022)

43 Huang, C., Fan, J., Li, W., Chen, X., Zhu, Q.: Reachnn: Reachability analysis of neural-network controlled systems. ACM Transactions on Embedded Computing Systems (TECS) **18**(5s), 1–22 (2019)

44 Hughes, A.: ChatGPT: Everything you need to know about OpenAI's GPT-4 tool. BBC Science Focus (2023)

45 Ivanov, R., Carpenter, T., Weimer, J., Alur, R., Pappas, G.J., Lee, I.: Verisig 2.0: Verification of neural network controllers using taylor model preconditioning. In: International Conference on Computer-Aided Verification (2021)

46 Ivanov, R., Carpenter, T.J., Weimer, J., Alur, R., Pappas, G.J., Lee, I.: Verifying the safety of autonomous systems with neural network controllers. ACM Trans. Embed. Comput. Syst. **20**(1) (Dec 2020). https://doi.org/10.1145/3419742

47 Ivanov, R., Weimer, J., Alur, R., Pappas, G.J., Lee, I.: Verisig: Verifying safety properties of hybrid systems with neural network controllers. In: International Conference on Hybrid Systems: Computation and Control. p. 169–178. HSCC, ACM (2019). https://doi.org/10.1145/3302504.3311806

48 Jia, K., Rinard, M.: Exploiting verified neural networks via floating point numerical error. In: International Static Analysis Symposium. pp. 191–205. Springer (2021)

49 Johnson, T.T., Lopez, D.M., Benet, L., Forets, M., Guadalupe, S., Schilling, C., Ivanov, R., Carpenter, T.J., Weimer, J., Lee, I.: Arch-comp21 category report: Artificial intelligence and neural network control systems (ainncs) for continuous and hybrid systems plants. In: Frehse, G., Althoff, M. (eds.) 8th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH21). EPiC Series in Computing, vol. 80, pp. 90–119. EasyChair (2021). https://doi.org/10.29007/kfk9, `https://easychair.org/publications/paper/Jq4h`

50 Johnson, T.T., Lopez, D.M., Musau, P., Tran, H.D., Botoeva, E., Leofante, F., Maleki, A., Sidrane, C., Fan, J., Huang, C.: Arch-comp20 category report: Artificial intelligence and neural network control systems (ainncs) for continuous and hybrid systems plants. In: Frehse, G., Althoff, M. (eds.) ARCH20. 7th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH20). EPiC Series in Computing, vol. 74, pp. 107–139. EasyChair (2020). https://doi.org/10.29007/9xgv, `https://easychair.org/publications/paper/Jvwg`

51 Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient smt solver for verifying deep neural networks. In: International Conference on Computer Aided Verification. pp. 97–117. Springer (2017)

52 Katz, G., Huang, D.A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljić, A., et al.: The Marabou framework for verification and analysis of deep neural

networks. In: International Conference on Computer Aided Verification. pp. 443–452. Springer (2019)

53 Kochdumper, N., Schilling, C., Althoff, M., Bak, S.: Open- and closed-loop neural network verification using polynomial zonotopes. In: NASA Formal Methods. pp. 16–36. Springer (2023)

54 Komendantskaya, E.: Proof-carrying neuro-symbolic code. In: Computability in Europe, CiE'25 (2025), `https://arxiv.org/abs/2504.12031`, to appear.

55 van Krieken, E., Acar, E., van Harmelen, F.: Analyzing differentiable fuzzy logic operators. Artificial Intelligence **302**, 103602 (2022). https://doi.org/10.1016/j.artint.2021.103602

56 Lopez, D.M., Althoff, M., Benet, L., Chen, X., Fan, J., Forets, M., Huang, C., Johnson, T.T., Ladner, T., Li, W., Schilling, C., Zhu, Q.: Arch-comp22 category report: Artificial intelligence and neural network control systems (ainncs) for continuous and hybrid systems plants. In: Frehse, G., Althoff, M., Schoitsch, E., Guiochet, J. (eds.) Proceedings of 9th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH22). EPiC Series in Computing, vol. 90, pp. 142–184. EasyChair (2022). https://doi.org/10.29007/wfgr, `https://easychair.org/publications/paper/C1J8`

57 Lopez, D.M., Althoff, M., Forets, M., Johnson, T.T., Ladner, T., Schilling, C.: Arch-comp23 category report: Artificial intelligence and neural network control systems (ainncs) for continuous and hybrid systems plants. In: Frehse, G., Althoff, M. (eds.) Proceedings of 10th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH23). EPiC Series in Computing, vol. 96, pp. 89–125. EasyChair (2023). https://doi.org/10.29007/x38n

58 Lopez, D.M., Althoff, M., Forets, M., Johnson, T.T., Ladner, T., Schilling, C.: ARCH-COMP23 category report: Artificial intelligence and neural network control systems (AINNCS) for continuous and hybrid systems plants. In: Frehse, G., Althoff, M. (eds.) Proceedings of 10th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH23), San Antonio, Texas, USA, May 9, 2023. EPiC Series in Computing, vol. 96, pp. 89–125. EasyChair (2023). https://doi.org/10.29007/X38N, `https://doi.org/10.29007/x38n`

59 Lopez, D.M., Althoff, M., Forets, M., Johnson, T.T., Ladner, T., Schilling, C.: Arch-comp23 category report: Artificial intelligence and neural network control systems (ainncs) for continuous and hybrid systems plants. In: Frehse, G., Althoff, M. (eds.) Proceedings of 10th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH23). EPiC Series in Computing, vol. 96, pp. 89–125. EasyChair (2023). https://doi.org/10.29007/x38n

60 Lopez, D.M., Choi, S.W., Tran, H.D., Johnson, T.T.: NNV 2.0: The neural network verification tool. In: Enea, C., Lal, A. (eds.) Computer Aided Verification. pp. 397–412. Springer Nature Switzerland, Cham (2023)

61 Lopez, D.M., Musau, P., Tran, H.D., Dutta, S., Carpenter, T.J., Ivanov, R., Johnson, T.T.: Arch-comp19 category report: Artificial intelligence and neural network control systems (ainncs) for continuous and hybrid systems plants. In: Frehse, G., Althoff, M. (eds.) ARCH19. 6th International Workshop on Applied Verification of Continuous and Hybrid Systems. EPiC Series in Computing, vol. 61, pp. 103–119. EasyChair (2019). https://doi.org/10.29007/rgv8, `https://easychair.org/publications/paper/BFKs`

62 Mandal, U., Amir, G., Wu, H., Daukantas, I., Newell, F.L., Ravaioli, U.J., Meng, B., Durling, M., Ganai, M., Shim, T., Katz, G., Barrett, C.W.: Formally verifying deep reinforcement learning controllers with lyapunov barrier certificates. CoRR **abs/2405.14058** (2024). https://doi.org/10.48550/ARXIV.2405.14058, `https://doi.org/10.48550/arXiv.2405.14058`

63 Müller, M.N., Makarchuk, G., Singh, G., Püschel, M., Vechev, M.: Prima: general and precise neural network certification via scalable convex hull approximations. Proceedings of the ACM on Programming Languages **6**(POPL), 1–33 (2022)

64 Namjoshi, K.S., Zuck, L.D.: Program correctness through self-certification. Commun. ACM **68**(2), 74–84 (Jan 2025). https://doi.org/10.1145/3689624, `https://doi.org/10.1145/3689624`

65 Norell, U.: Dependently typed programming in Agda. In: International school on advanced functional programming. pp. 230–266. Springer (2008)

66 Păsăreanu, C.S., Mangal, R., Gopinath, D., Getir Yaman, S., Imrie, C., Calinescu, R., Yu, H.: Closed-loop analysis of vision-based autonomous systems: A case study. In: International Conference on Computer Aided Verification. pp. 289–303. Springer (2023)

67 Seshia, S.A., Sadigh, D., Sastry, S.S.: Toward verified artificial intelligence. Communications of the ACM **65**(7), 46–55 (2022)

68 Seshia, S.A., Sadigh, D., Sastry, S.S.: Toward verified artificial intelligence. Commun. ACM **65**(7), 46–55 (Jun 2022). https://doi.org/10.1145/3503914

69 Shi, G., Shi, X., O'Connell, M., Yu, R., Azizzadenesheli, K., Anandkumar, A., Yue, Y., Chung, S.J.: Neural lander: Stable drone landing control using learned dynamics. In: 2019 international conference on robotics and automation (icra). pp. 9784–9790. IEEE (2019)

70 Shriver, D., Elbaum, S., Dwyer, M.B.: DNNV: A Framework for Deep Neural Network Verification. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification. pp. 137–150. Springer International Publishing, Cham (2021)

71 Sidrane, C., Kochenderfer, M.J.: OVERT: Verification of nonlinear dynamical systems with neural network controllers via overapproximation. Safe Machine Learning workshop at ICLR (2019)

72 Slusarz, N., Komendantskaya, E., Daggitt, M.L., Stewart, R.J., Stark, K.: Logic of differentiable logics: Towards a uniform semantics of DL. In: LPAR-24: The International Conference on Logic for Programming, Artificial Intelligence and Reasoning (2023)

73 Smart, J.: Implementation of a Rocq backend to the vehicle neural network specfication language. Tech. rep., BSc Dissertation, University of Southampton (2025)

74 Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., Fergus, R.: Intriguing properties of neural networks. In: International Conference on Learning Representations (2013), `http://arxiv.org/abs/1312.6199`

75 Teuber, S., Mitsch, S., Platzer, A.: Provably safe neural network controllers via differential dynamic logic. CoRR **abs/2402.10998** (2024). https://doi.org/10.48550/ARXIV.2402.10998, `https://doi.org/10.48550/arXiv.2402.10998`

76 Tran, H.D., Xiang, W., Johnson, T.T.: Verification approaches for learning-enabled autonomous cyber–physical systems. IEEE Design & Test **39**(1), 24–34 (2022). https://doi.org/10.1109/MDAT.2020.3015712

77 Tran, H.D., Yang, X., Lopez, D.M., Musau, P., Nguyen, L.V., Xiang, W., Bak, S., Johnson, T.T.: NNV: The neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. In: 32nd International Conference on Computer-Aided Verification (CAV'20) (7 2020)

78 Wang, S., Zhang, H., Xu, K., Lin, X., Jana, S., Hsieh, C.J., Kolter, J.Z.: Beta-CROWN: Efficient bound propagation with per-neuron split constraints for complete and incomplete neural network verification. Advances in Neural Information Processing Systems **34** (2021)

79 Wüthrich, M.V.: Bias regularization in neural network models for general insurance pricing. European Actuarial Journal **10**(1), 179–202 (2020)

80 Yuan, L., Chen, D., Chen, Y.L., Codella, N., Dai, X., Gao, J., Hu, H., Huang, X., Li, B., Li, C., Liu, C., Liu, M., Liu, Z., Lu, Y., Shi, Y., Wang, L., Wang, J., Xiao, B., Xiao, Z., Yang, J., Zeng, M., Zhou, L., Zhang, P.: Florence: A new foundation model for computer vision (2021)

81 Zhang, H., Wang, S., Xu, K., Li, L., Li, B., Jana, S., Hsieh, C.J., Kolter, J.Z.: General cutting planes for bound-propagation-based neural network verification. Advances in Neural Information Processing Systems (2022)