Formal Verification of Neural-Cyber-Physical systems

Formal Methods in Australia and New Zealand

Matthew Daggitt (University of Western Australia)

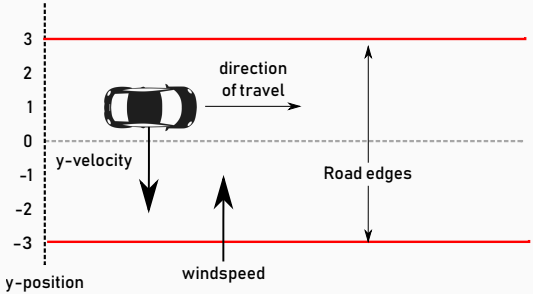
5th of June, 2026

Contributors

- Ekaterina Komendanskaya (University of Southampton)
- Bob Atkey (University of Strathclyde)
- Wen Kokke (Well-Typed)
- Alessandro Bruni, Gusts Grinbergs (IT-University of Copenhagen)
- Samuel Teuber (Karlsruhe Institute of Technology)
- Grant Passmore (Imandra)
- Many more...

Motivating example

Example: staying on the road



- Strong unpredictable cross-wind
- Noisy sensor reading of position

Goal: keep the car on the road!

Example: staying on the road

$\text{controller} : \mathbb{Q} \to \mathbb{Q} \to \mathbb{Q}$

Inputs:

- Current sensor reading
- Previous sensor reading

Outputs:

- Change in velocity

Example: staying on the road

```
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
```

Example: staying on the road

```
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
```

Example: staying on the road

Theorem

Assuming that the wind-speed can shift by no more than 1 per unit time and that the sensor is never off by more than 0.25 units then the car will never leave the road.

Example: staying on the road

The final desired safety property can be encoded as:

$$\text{ValidObservation} : \text{Observation} \to \text{Set}$$
$$\text{ValidObservation}\ o = |\ \text{sensorError}\ o\ | \leq 0.25 \times |\ \text{windShift}\ o\ | \leq 1$$

$$\text{OnRoad} : \text{State} \to \text{Set}$$
$$\text{OnRoad}\ s = |\ \text{position}\ s\ | \leq 3$$

$$\text{finalState-onRoad} : \forall\ xs \to \text{All ValidObservation}\ xs \to \text{OnRoad}\ (\text{finalState}\ xs)$$

Question: How can we prove the required result when:

$$\text{controller} : \mathbb{Q} \to \mathbb{Q} \to \mathbb{Q}$$

is implemented as a neural network?

Solving neuro-cyber-physical verification problems

Decomposing the problem

A NCPS can be represented as $W(S(N))$ where:

- W is a symbolic model of the world in which the agent acts.
- S is a symbolic model of the conventional software components.
- N is the sub-symbolic neural network controller.

The verification goal is then to establish a safety property $\Phi(W(S(N)))$,

Neural Network Verification is a Programming Language Challenge,
https://arxiv.org/abs/2501.05867 (ESOP 2025)

Modelling an AI agent in an environment

The dominant automated approach is to prove $\Phi(W(S(N)))$ by:

1. encoding $W(S(N))$ as a symbolic n-step transition system.
2. use automated reachability analysis to prove safety properties.

Many toolboxes exist compete annually in ARCHCOMP, e.g. `PolarExpress`, `CORA`, `NNV`.

Downsides:

- Only support finite time horizons, i.e. n steps in the future.
- Performance decreases rapidly as n and the complexity of W and S grows.

An alternative compositional approach

In order to prove infinite-horizon safety properties, we need to exploit the structure of the problem!

From Φ, derive a specification ψ of the neural network such that the proof of $\Phi(W(S(N)))$ can be decomposed as follows:

- $\forall f.\psi(f) \Rightarrow \Phi(W(S(f)))$
- $\psi(N)$

Example: staying on the road

Going back to the case study the desired result:

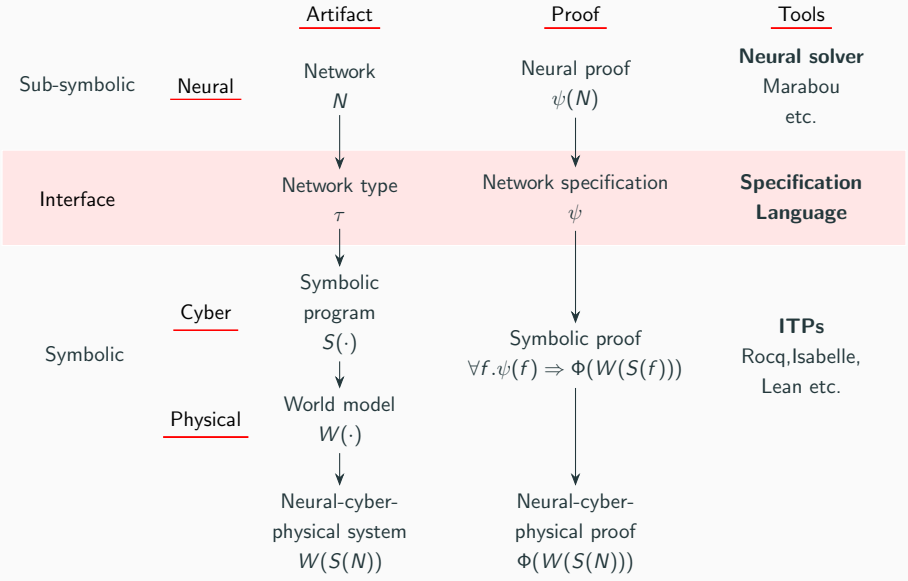
$$\text{finalState-onRoad} : \forall\ xs \rightarrow \text{All ValidObservation } xs \rightarrow \text{OnRoad } (\text{finalState } xs)$$

can be proved by induction if the following lemma holds:

$$\begin{aligned}\text{controller-lemma} : \ & \forall\ x\ y \rightarrow |\ x\ | \leq 3.25 \rightarrow |\ y\ | \leq 3.25 \rightarrow \\ & |\ \text{controller } x\ y + 2 * x - y\ | < 1.25\end{aligned}$$

This is our ψ!

System decomposition



Proving network property $\psi(N)$

Around 2016, the automatic theorem prover (ATP) community collectively started working on this problem.

A range of domain-specific neural network solvers now exist:

- Marabou (SMT technology)
- α-β-Crown (abstract interpretation + MILP)
- Verisig (interval arithmetic)

with many others...

Proving network property $\psi(N)$

The verifiers can be thought of as specialised SMT solvers (with domain-specific limitations).

Given property ψ represented as a set of constraints over variables representing the inputs and outputs, they find a satisfying assignment of input variables.

Example 1 Example 2

These live in the sub-symbolic world, and nobody wants to write these by hand!

Specification language

Clearly we need a specification language to write down ψ and compile it to VNN-LIB queries.

The obvious approach:

- Encode network N as a symbolic function in your ITP.
- Write down ψ in a DSL embedded in the ITP (e.g. Rocq, Lean, Isabelle/HOL).
- Write a compiler to VNN-LIB.
- Verify $\psi(N)$ as a tactic that uses the external solver.

Specification language

Clearly we need a specification language to write down ψ and compile it to VNN-LIB queries.

The obvious approach:

- Encode network N as a symbolic function in your ITP.
- Write down ψ in a DSL embedded in the ITP (e.g. Rocq, Lean, Isabelle/HOL).
- Write a compiler to VNN-LIB.
- Verify $\psi(N)$ as a tactic that uses the external solver.

Unfortunately, this doesn't work in practice...

First Set of Problems

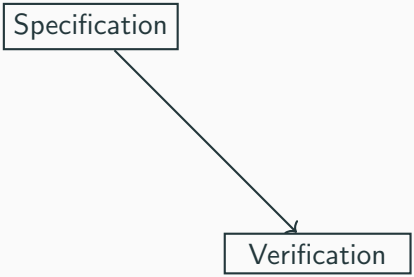
The first set of problems:

1. **Even writing down the network N may crash your ITP.** Even small neural networks have tens of millions of parameters and PyTorch has 3500+ different operator types.
2. **ITPs are not performant enough for compilation.** If the inputs to N are high-dimensional (e.g. images) then ψ will contain millions of variables and constraints.
3. **ITPs don't support deployment.** The final verified network often needs to be deployed to custom hardware (e.g. ASICs, FPGAs).

The Broader Picture

Specification

The Broader Picture

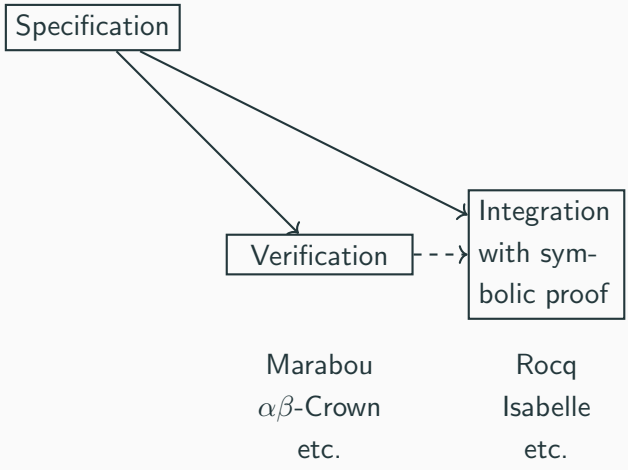


Marabou

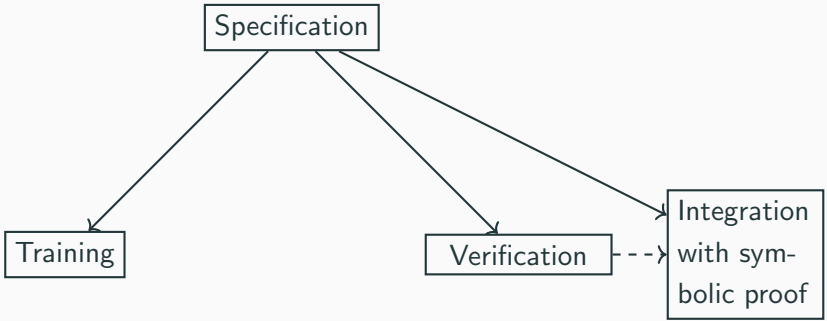
$\alpha\beta$-Crown

etc.

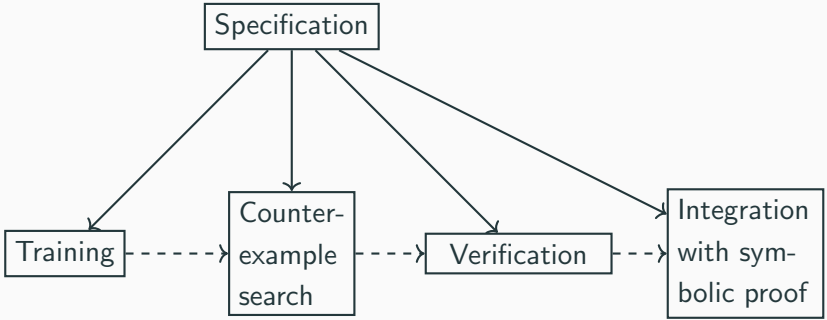
The Broader Picture



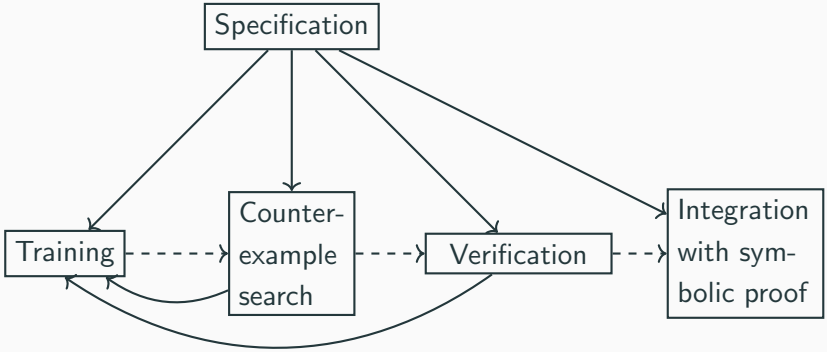
The Broader Picture



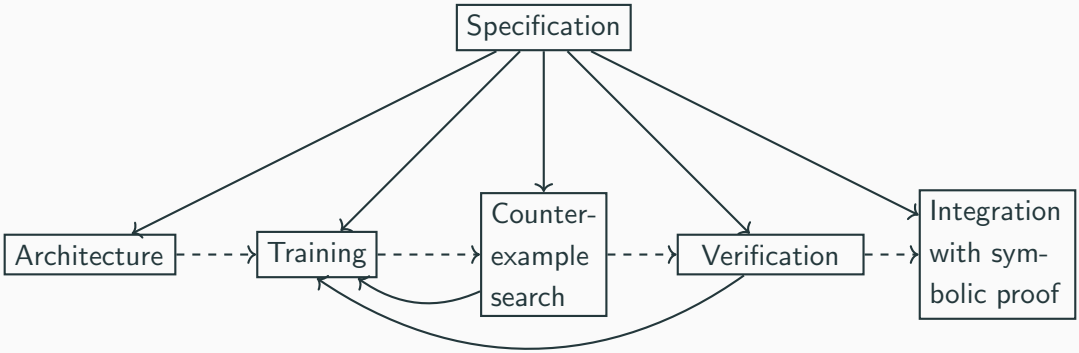
The Broader Picture



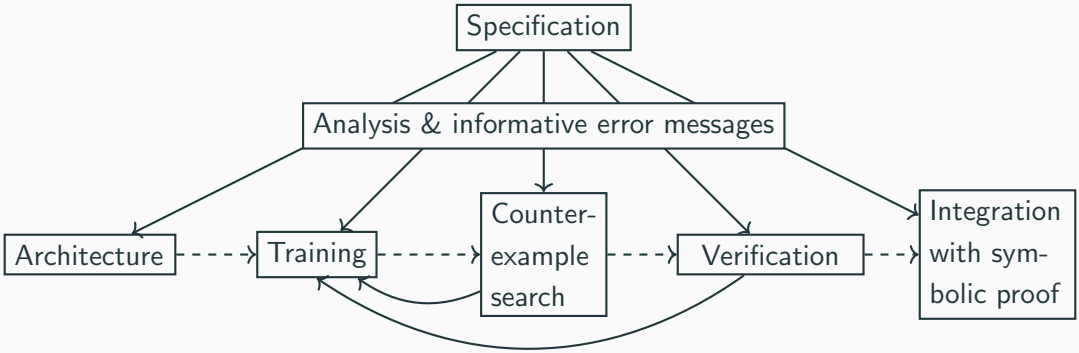
The Broader Picture



The Broader Picture



The Broader Picture



Second Set of Problems

The second set of problems:

4. **ITPs have no integration with training libraries.** Reimplementing industry-grade libraries such as PyTorch natively in an ITP is infeasible.
5. **Any solution would be ITP-specific.** Even if 1000s of hours were spent building this infrastructure, it would be limited to a single ITP.
6. **Unfriendly to ML practitioners.** Training and counter-example search would be useful to general ML practioners. They are unlikely to want to work in an ITP!

Solution

The solution is to have an external specification language that can:

1. integrate with ML libraries such as PyTorch and Tensorflow.
2. compile to VNN-LIB queries for verification.
3. compile to ITP code for integration with symbolic proofs.

The Vehicle framework

Vehicle

Vehicle

Vehicle has a high-level, dependently-typed specification language with native support for tensors, networks, datasets and higher-order functions.

Examples include:

- Wind Controller (the example in this talk)
- Adversarial Robustness.
- Drone Collision Avoidance.

Vehicle types and semantics

The language has a standard dependent-type system, and for any well-typed expression we can define a compositional denotational semantics, $[\![\cdot]\!]$, e.g.:

$$
\begin{aligned}
[\![\mathit{Bool}]\!] &= \mathbb{B} \\
[\![\mathit{True}]\!] &= \top \\
[\![\mathit{False}]\!] &= \bot \\
[\![e_1 \text{ or } e_2]\!] &= [\![e_1]\!] \vee [\![e_2]\!] \\
[\![e_1 \text{ and } e_2]\!] &= [\![e_1]\!] \wedge [\![e_2]\!] \\
[\![\text{not } e]\!] &= \neg [\![e]\!] \\
\ldots &= \ldots
\end{aligned}
$$

Loss backend

Loss function backend

The semantics of a @property p in a specification parameterised by a @network f can be seen as a function:

$$[\![p]\!] : (\texttt{Tensor Real } m \to \texttt{Tensor Real } n) \to \mathbb{B}$$

Suppose we could can turn this into a function $[\![p]\!]_l$ of type:

$$[\![p]\!]_l : (\texttt{Tensor Real } m \to \texttt{Tensor Real } n) \to \mathbb{R}$$

where the output represents "how true" the property p is.

If $[\![p]\!]_l$ is differentiable, then we could use gradient descent to optimise for f via $\frac{\partial [\![p]\!]_l(f)}{\partial f}$ ...

Differentiable Logics

The key idea is to define an alternative compositional, denotational semantics $[\![\cdot]\!]_I$, known as a *Differentiable Logic*, for boolean expressions in the language, e.g.

$$
\begin{aligned}
[\![\mathit{Bool}]\!]_I &= [0,1] \subset \mathbb{R} \\
[\![\mathit{True}]\!]_I &= 0 \\
[\![\mathit{False}]\!]_I &= 1 \\
[\![e_1 \texttt{ or } e_2]\!]_I &= [\![e_1]\!]_I [\![e_2]\!]_I \\
[\![e_1 \texttt{ and } e_2]\!]_I &= [\![e_1]\!]_I + [\![e_2]\!]_I - [\![e_1]\!]_I [\![e_2]\!]_I \\
[\![\texttt{not } e]\!]_I &= 1 - [\![e]\!]_I \\
\ldots &= \ldots
\end{aligned}
$$

Differentiable Logic research questions

Research questions we are currently tackling:

1. How do we trade-off logical properties of the semantics with the practical performance of the resulting loss function?
2. What should soundness and completeness of these alternative semantics mean?
3. What is the right semantics for quantifiers?

Verifier backend

Marabou

What we want to prove:

$$\text{controller-lemma} : \forall\ x\ y \rightarrow |\ x\ | \leq 3.25 \rightarrow |\ y\ | \leq 3.25 \rightarrow |\ \text{controller}\ x\ y + 2 * x - y\ | < 1.25$$

Equi-satisfiable Marabou queries:

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

Query 1

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

Query 2

Let's try it out!

Compiling to verifier backends

We need to compile Boolean expressions down to an equi-satisfiable set of queries.

While compiling we need to:

1. Change the model of the network from a function to a relation.
2. Move quantified variables from the problem-space to the embedding-space.
3. Detect and explain non-linearities, alternating quantifiers.
4. Ensure compilation time is linear the input/output dimensions of network.

Surprisingly challenging! We currently have an partially algorithm...

Efficient compilation of expressive problem space specifications to neural network solvers, Daggitt, Kokke, Atkey, https://arxiv.org/abs/2402.01353

Compiling Higher-Order Specifications to SMT Solvers: How to Deal with Rejection Constructively, Daggitt, Atkey, Kokke, Komendantskaya, Arnaboldi, CPP2023.

ITP backend

Interactive theorem provers

Many fantastic interactive theorem prover (ITP) systems out there:

- General, self-contained systems: Rocq, Agda, etc.
- Specialised, self-contained systems: KeYmaera X, etc.
- Future systems based on emerging theory, e.g. probabilistic programming.

Each has it's own speciality libraries.

Adding Vehicle support for a new ITP takes 800 lines of fairly formulaic Haskell + 200 lines of ITP code.

ITP backend challenges

There are two major challenges in supporting export to ITPs:

- Some of Vehicle's 'bool' expressions are decidable within the ITP, some are not.
- How to invalidate the proof when the neural network artefact changes on disk.

Compositional Neural-Cyber-Physical System Verification in the Interactive Theorem Prover of Your Choice, Daggitt, Komendantskaya, Sirman, Bruni, Teuber, Smart, Passmore. To appear ICFP 2026.

Vehicle uptake

Community engagement

- 22 contributors from 7 universities.
- Used by Schlumeger Research Division, Cambridge.
- Used by Imandra, a for-profit automated theorem prover in the states.
- Used to teach advanced verification courses at University of Edinburgh.

Real world uses:

- Robustness of Network Intrusion Detection System
- Safety of drug-delivery system.
- Drone controller.
- Others (unknown).

Conclusions

Conclusions

Vehicle allows users to write formal specifications for an AI agent and then facilitates training, verification and integration with upstream symbolic proofs.

It has strong theoretical foundations based on:

- Dependent-type theory.
- Standard and novel denotational semantics.

while also having a strong emphasis on usability:

- Intuitive specification language.
- Excellent error messages.
- Many performance issues overcome.
- Detailed documentation.

Trying Vehicle out

You can try it out:

```
pip install vehicle-lang maraboupy
```

Source code available at:

https://github.com/vehicle-lang/vehicle

Collaborations

Moving forwards:

- There's a variety of interesting theoretical problems to be solved in every backend.
- We're keen to find more real-world problems.
- Eager to collaborate if people like any of these ideas or have others!

Appendices

Counter-example search

Given a property p of either of the following forms:

$$\texttt{forall}\,(\texttt{x} : \tau).\ e$$

$$\texttt{exists}\,(\texttt{x} : \tau).\ e$$

we would like to find a cheap procedure to find an instantiation for x that either acts as a witness or a counter-example for the property.

Counter-example search via loss functions

One approach is to again use differential logic...

Before, we were training the network to satisfy property p by using gradient descent to optimise for f via $\frac{\partial [\![p]\!]_I(f)}{\partial f}$.

However, if we remove sampling and instead just treat the quantified variable x as a free variable we could instead use gradient descent to optimise for x via $\frac{\partial [\![p]\!]_I(f,x)}{\partial x}$.

Soon to be implemented...