Developing programming language support for end-to-end verification of neural AI agents

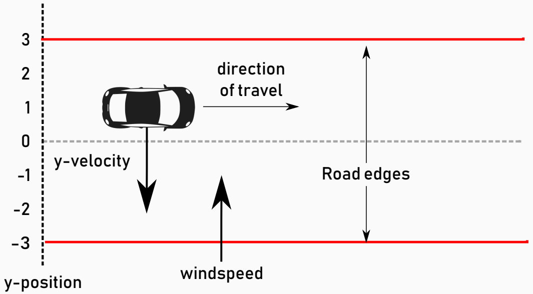
Topos Institute seminar

Matthew Daggitt (University of Western Australia)
Wen Kokke & Bob Atkey (University of Strathclyde)
Ekaterina Komendantskya (Southampton University)

1st of May, 2025

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 then the car will never leave the road.

Example: staying on the road

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

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

Example: staying on the road

The desired result:

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

Example: staying on the road

The desired result:

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

Proof is inductive and involves algebraic manipulation but relies on the lemma:

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

Example: staying on the road

So we have:

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

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

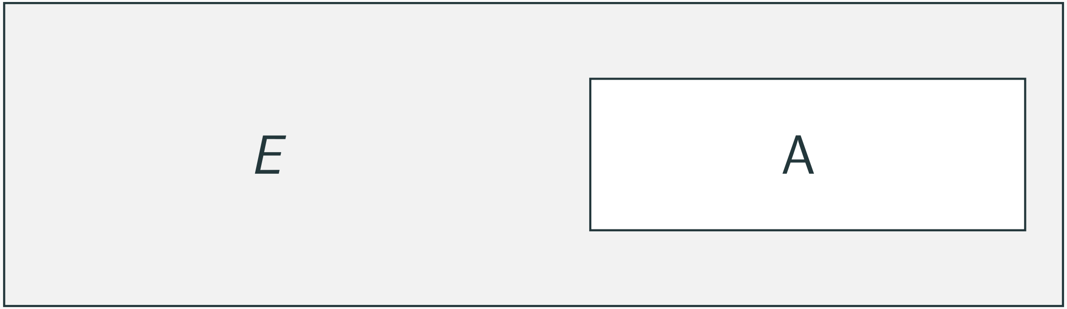
How can we implement the controller as a neural network and prove the required result?

Our contributions

1. A general decomposition of the AI agent verification problem into parts.
2. Our tool "*Vehicle*" which facilitates this decomposition.

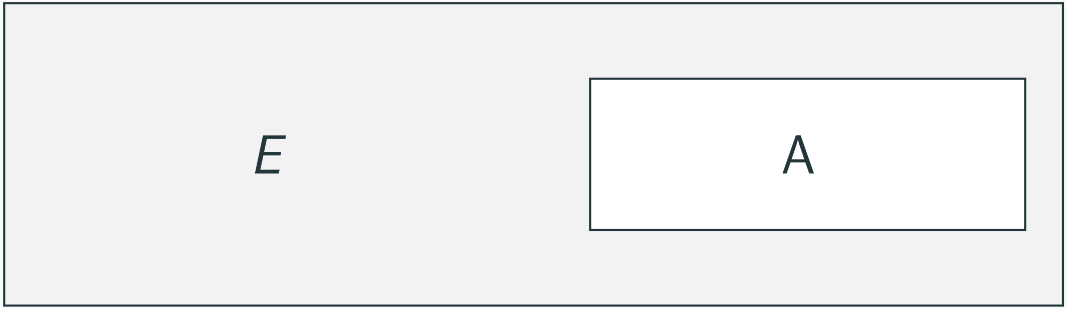
Decomposing the AI agent verification problem

Modelling an AI agent in an environment



An agent A is acting in an environment E.

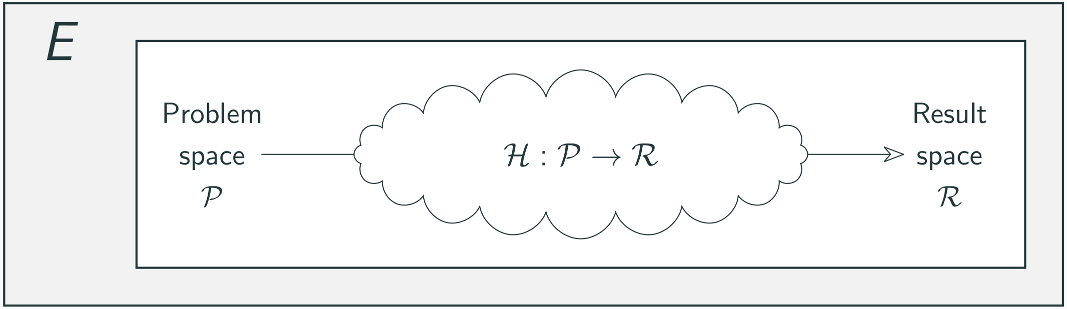
Modelling an AI agent in an environment



We have a safety property P, and we want to prove $P(E, A)$, e.g.

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

Modelling an AI agent in an environment

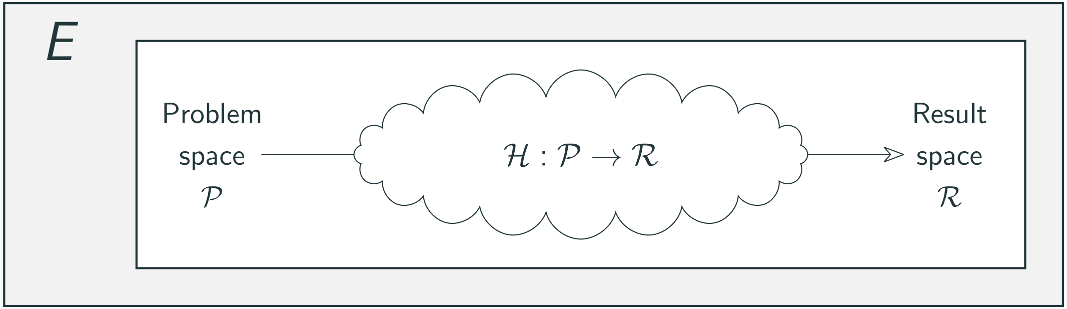

E
Problem space $\mathcal{P}$
$\mathcal{H} : \mathcal{P} \to \mathcal{R}$
Result space $\mathcal{R}$

We believe there exists a solution $\mathcal{H}$ such that $P(E, \mathcal{H})$

e.g. $\mathcal{P} = \mathbb{R}^2$ (sensor readings), $\mathcal{R} = \mathbb{R}$ (change in velocity)

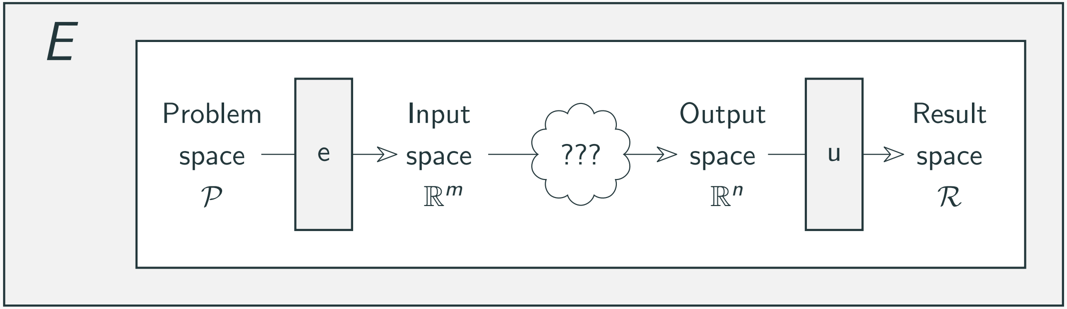
Problem and result spaces are <u>semantically rich</u>.

Modelling an AI agent in an environment


E
Problem space $\mathcal{P}$
$\mathcal{H} : \mathcal{P} \to \mathcal{R}$
Result space $\mathcal{R}$

Goal: approximate $\mathcal{H}$ using machine learning

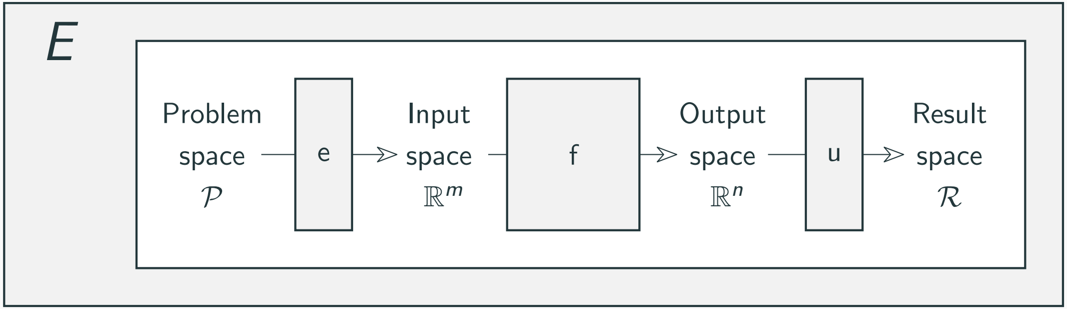
Modelling an AI agent in an environment


E
Problem space $\mathcal{P}$
e
Input space $\mathbb{R}^m$
???
Output space $\mathbb{R}^n$
u
Result space $\mathcal{R}$

Need an *embedding function* e and *unembedding function* u.

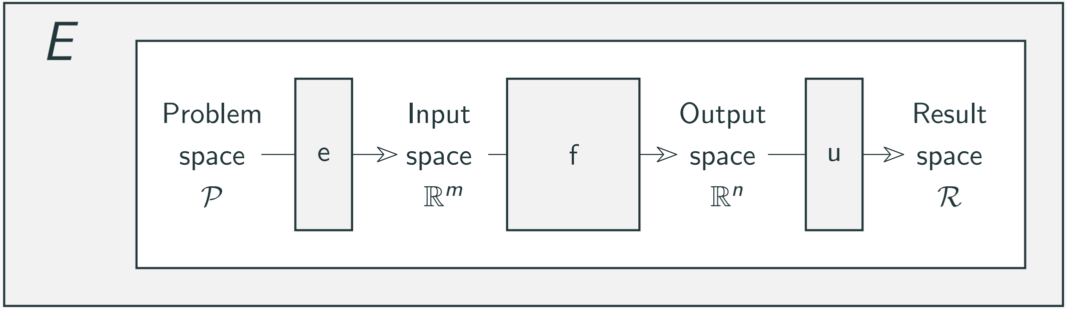
Input and output spaces are semantically meaningless.

Modelling an AI agent in an environment


E
Problem space $\mathcal{P}$
e
Input space $\mathbb{R}^m$
f
Output space $\mathbb{R}^n$
u
Result space $\mathcal{R}$

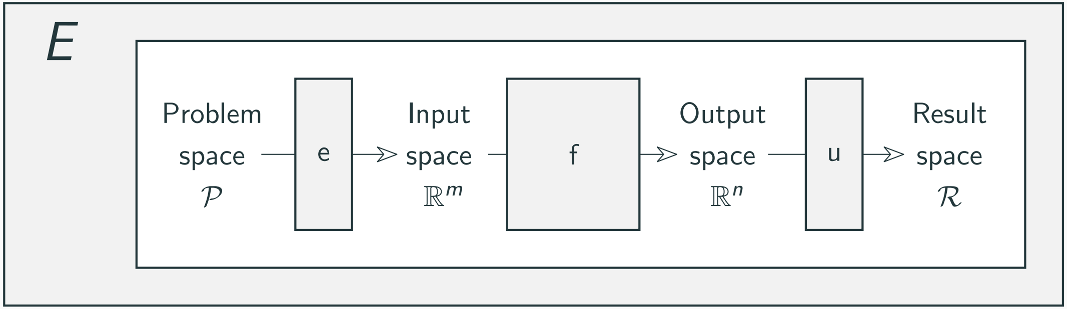
Then train a machine learning system f.

Modelling an AI agent in an environment


E
Problem space $\mathcal{P}$
e
Input space $\mathbb{R}^m$
f
Output space $\mathbb{R}^n$
u
Result space $\mathcal{R}$

Goal: prove $P(E, u \circ f \circ e)$

Modelling an AI agent in an environment



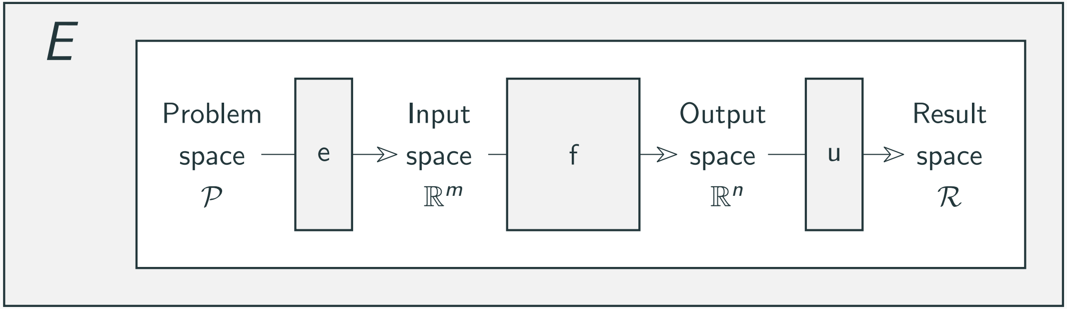
E
Problem space $\mathcal{P}$
e
Input space $\mathbb{R}^m$
f
Output space $\mathbb{R}^n$
u
Result space $\mathcal{R}$

Goal: find a property Q such that:

i) $\forall h. Q(h) \Rightarrow P(E, h)$ ii) $Q(u \circ f \circ e)$

controller-lemma : $\forall$ *x y* → ... → | controller *x y* + *2* * *x* - *y* | < *1.25*

Modelling an AI agent in an environment


E
Problem space $\mathcal{P}$
e
Input space $\mathbb{R}^m$
f
Output space $\mathbb{R}^n$
u
Result space $\mathcal{R}$

Goal: find properties Q and R such that:

i) $\forall h. Q(h) \Rightarrow P(E, h)$ ii) $R(f) \Rightarrow Q(u \circ f \circ e)$ iii) $R(f)$

Proof strategy

Therefore to prove $P(E, u \circ f \circ e)$:

1. Find problem space property $Q : (\mathcal{P} \to \mathcal{R}) \to \mathbb{B}$ 2. Prove $\forall h. Q(h) \Rightarrow P(E, h)$	Environment proofs
3. Find input space property $R : (\mathbb{R}^m \to \mathbb{R}^n) \to \mathbb{B}$ 4. Prove $R(f) \Rightarrow Q(u \circ f \circ e)$	Embedding proofs
5. Prove $R(f)$	Network proofs

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

Proof strategy

Therefore to prove $P(E, u \circ f \circ e)$:

1. Find problem space property $Q : (\mathcal{P} \to \mathcal{R}) \to \mathbb{B}$
2. Prove $\forall h. Q(h) \Rightarrow P(E, h)$

Environment proofs - ITPs

---

3. Find input space property $R : (\mathbb{R}^m \to \mathbb{R}^n) \to \mathbb{B}$
4. Prove $R(f) \Rightarrow Q(u \circ f \circ e)$

Embedding proofs - ???

---

5. Prove $R(f)$

Network proofs - ???

Proof strategy

Therefore to prove $P(E, u \circ f \circ e)$:

1. Find problem space property $Q : (\mathcal{P} \to \mathcal{R}) \to \mathbb{B}$
2. Prove $\forall h. Q(h) \Rightarrow P(E, h)$

Environment proofs - ITPs

---

3. Find input space property $R : (\mathbb{R}^m \to \mathbb{R}^n) \to \mathbb{B}$
4. Prove $R(f) \Rightarrow Q(u \circ f \circ e)$

Embedding proofs - ???

---

5. Prove $R(f)$

Network proofs - ???

Step 5. Prove $R(f)$

Infeasible to do in an ITP:

- Neural networks are massive.
- No semantically meaningful subcomponents.
- Even writing down R is painful!

Step 5. Prove $R(f)$

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

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

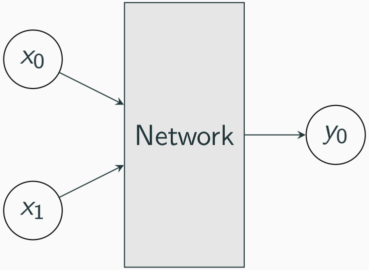
- Marabou (SMT technology)
- α-β-Crown (abstract interpretation + MILP)
- Verisig (interval arithmetic)

with many others...

Step 5. Prove $R(f)$

The verifiers can be thought of as specialised SMT solvers.

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



Examples of R: Example 1 Example 2

Proof strategy

Therefore to prove $P(E(u \circ f \circ e))$:

1. Find problem space property $Q : (\mathcal{P} \to \mathcal{R}) \to \mathbb{B}$
2. Prove $\forall h. Q(h) \Rightarrow P(E(h))$

Environment proofs - ITPs

---

3. Find input space property $R : (\mathbb{R}^m \to \mathbb{R}^n) \to \mathbb{B}$
4. Prove $R(f) \Rightarrow Q(u \circ f \circ e)$

Embedding proofs - ???

---

5. Prove $R(f)$

Network proofs - ATPs

Step 3. Find $R : (\mathbb{R}^m \to \mathbb{R}^n) - > \mathbb{B}$

R has no semantic meaning... which strongly suggests that it should be automatically derived from Q and the (un)embedding functions e and u.

Step 4. Prove $R(f) \Rightarrow Q(u \circ f \circ e)$

ATPs are a bad match:

- highly specialised for reasoning about neural network architectures.
- embedding functions contain arbitrary computation and problem space representations.

ITPs are a bad match:

- given that user doesn't want to even write down R...
- ... they definitely don't want to have to directly reason about it!

Embedding gap

Therefore to prove $P(E(u \circ f \circ e))$:

1. Find problem space property $Q : (P \to R) \to \mathbb{B}$
2. Prove $\forall h. Q(h) \Rightarrow P(E(h))$

System proofs - ITPs

---

3. Find input space property $R : (\mathbb{R}^m \to \mathbb{R}^n) \to \mathbb{B}$
4. Prove $R(f) \Rightarrow Q(u \circ f \circ e)$

Embedding proofs - ???

---

5. Prove $R(f)$

Network proofs - ATPs

Goal

$\boxed{\text{Specification (Q)}}$

Goal

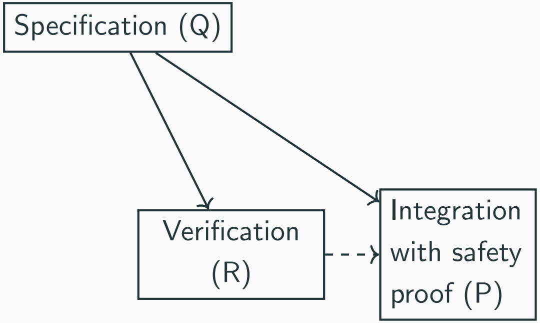
$$\begin{tikzcd}
\boxed{\text{Specification (Q)}} \arrow[dr] & \\
& \boxed{\text{Verification (R)}}
\end{tikzcd}$$

Marabou

$\alpha\beta$-Crown

etc.

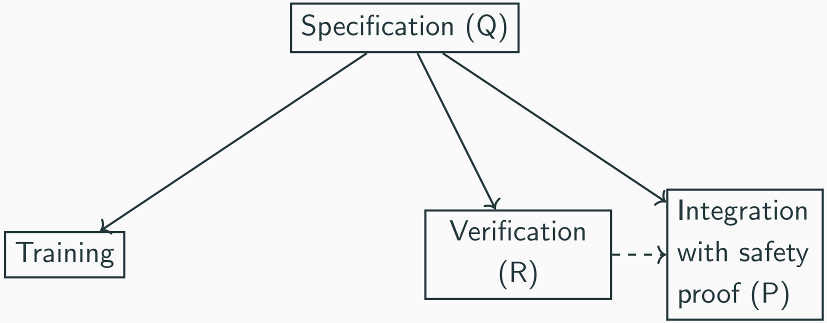
Goal



Marabou
$\alpha\beta$-Crown
etc.

Rocq
Agda
etc.

Goal

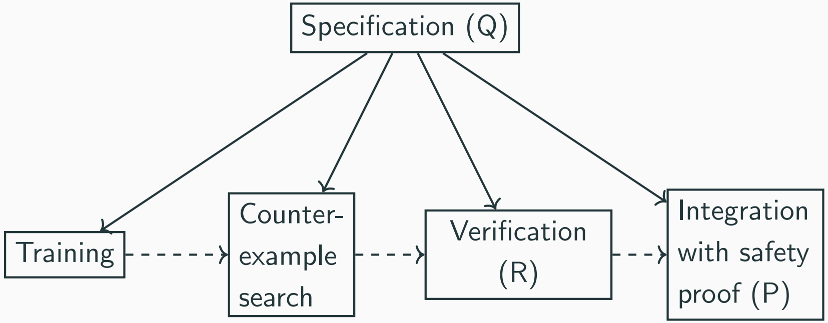


Tensorflow
PyTorch
etc.

Marabou
$\alpha\beta$-Crown
etc.

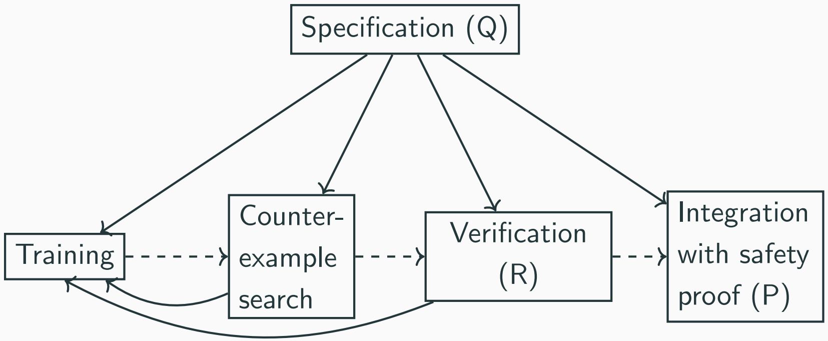
Rocq
Agda
etc.

Goal



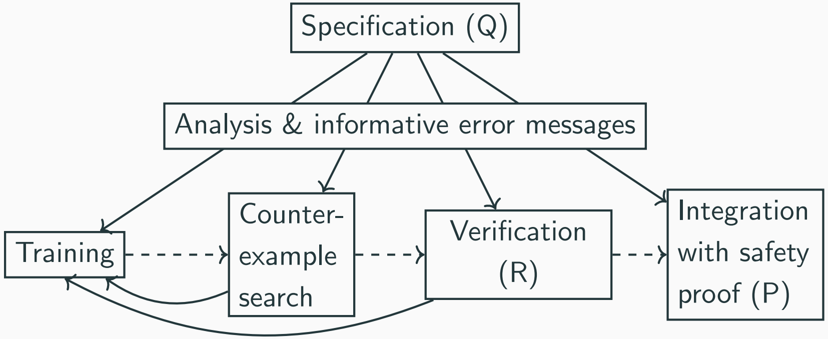
Tensorflow	PGD	Marabou	Rocq
PyTorch	FGSM	$\alpha\beta$-Crown	Agda
etc.	etc.	etc.	etc.

Goal



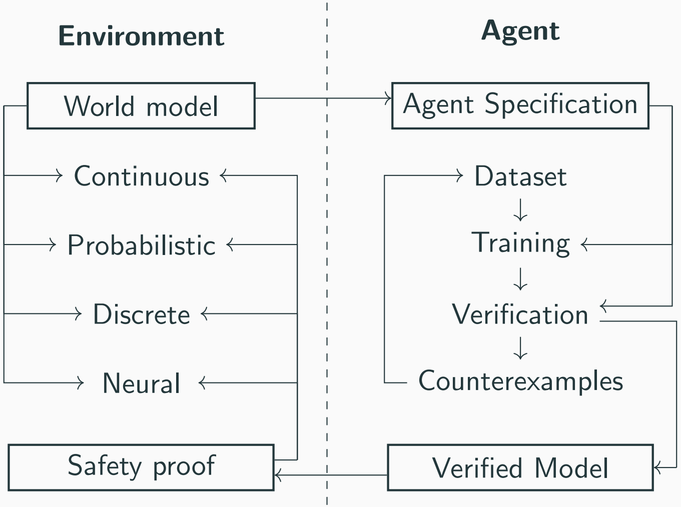
Tensorflow	PGD	Marabou	Rocq
PyTorch	FGSM	$\alpha\beta$-Crown	Agda
etc.	etc.	etc.	etc.

Goal

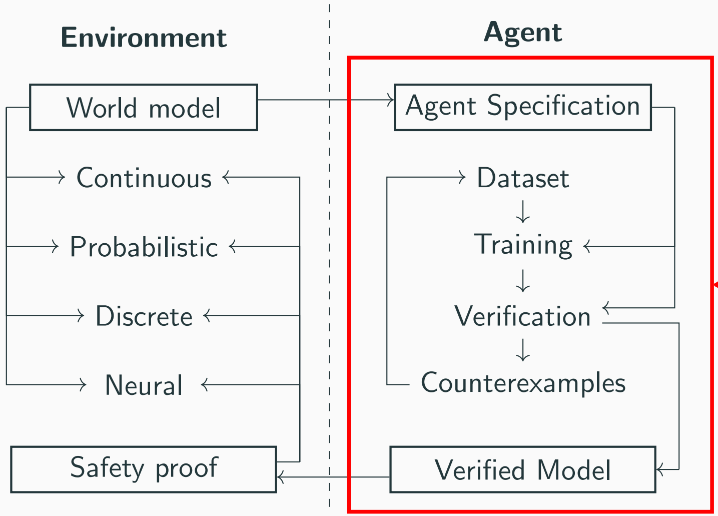

Specification (Q)
Analysis & informative error messages
Training
Counter-example search
Verification (R)
Integration with safety proof (P)

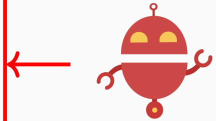
Tensorflow	PGD	Marabou	Rocq
PyTorch	FGSM	$\alpha\beta$-Crown	Agda
etc.	etc.	etc.	etc.

Overall picture



Overall picture





Our Vehicle tool

Vehicle

Vehicle

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

Vehicle specifications can currently be compiled to:

- Loss functions for training.
- Marabou queries for verification.
- Agda modules for integration with the full safety proofs.

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.:

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

Training backend

Loss function backend

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

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

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

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

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 $\frac{\partial [\![p]\!]_l(f)}{\partial f}$...

Loss functions

The key idea is to define an alternative compositional, denotational semantics $[\![\cdot]\!]_l$, for boolean expressions in the language, e.g.

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

Differentiable logics

There are many different ways to translate the boolean operations. Each one is known as *differentiable logic*, e.g.

- **Product logic** - "Analyzing differentiable fuzzy logic" Krieken et al. (2020)
- **DL2** - "Training and querying neural networks with logic" Fischer et al. (2019)
- **Quantative Logic** - "On Quantifiers for Quantitative Reasoning" Cappuci (2024)

Regardless of the differentiable logic, you would hope that the translation was *sound*, i.e. for any well-typed expression $e : \mathit{Bool}$:

$$\forall e.\ (\llbracket e \rrbracket_I = \llbracket \mathit{True} \rrbracket_I) \Rightarrow (\llbracket e \rrbracket = \mathit{True})$$

Quantifier semantics

Some logics (e.g. DL2) define a non-compositional semantics for a single outermost quantifier...

Does there exist an executable compositional semantics for `forall` and `exists` that preserves differentiability and some notion of soundness?

$$[\![\texttt{forall}\,(\texttt{x} : \tau)\texttt{.}\; e]\!]_I = ??? \qquad [\![\texttt{exists}\,(\texttt{x} : \tau)\texttt{.}\; e]\!]_I = ???$$

Implementation of quantifier semantics

We've implemented this as sampling:

$$[\![\texttt{forall}\,(\texttt{x} : \tau).\; e]\!]_I = [\![\bigwedge_{i=0}^{n}]\!]_I\,([\![e]\!]_I(x_i)) \qquad \text{where } x_i \sim \mathcal{D}([\![\tau]\!]_I)$$

"Logic of Differentiable Logics: Towards a Uniform Semantics of DL", Slusarz, Komendantskaya, Daggitt, Stewart, Stark, LPAR 2023

Verifier backend

Marabou

What we want to prove:

controller-lemma : $\forall$ *x y* $\rightarrow$ | *x* | $\leq$ *3.25* $\rightarrow$ | *y* | $\leq$ *3.25* $\rightarrow$
| 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

To use the verifiers we need to compile a well-typed Vehicle expression e of type Bool 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. Remove deeply embedded if-statements in higher-order functions.
4. Ensure compilation time is linear the input/output dimensions of network.

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

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

Good error messages for SMT-based solvers

An important usability concern is to ensure good error reporting!

Marabou only supports linear, first-order specifications but Vehicle is a much more expressive language!

Therefore we need to reject some specifications in a way that provides useful explanations to the user.

This is a problem common to many specification languages that integrate ATPs (e.g. Liquid Haskell, Daphny, Isabelle)

Error messages for SMT-based solvers

We can reuse the same dependent-type checking algorithm that Vehicle uses to type-check the original specification, to over-approximate membership of the set of first-order, linear properties.

Compiling Higher-Order Specifications to SMT Solvers: How to Deal with Rejection Constructively, Daggitt, Atkey, Kokke, Komendantskaya, Arnaboldi, CPP 2023, https://laiv.uk/wp-content/uploads/2022/12/vehicle.pdf

ITP backend

Interactive theorem provers

The final challenge is export the verified specifications to systems for building world-models for the environment.

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. double categorical systems theory, probabilistic programming.

Vehicle currently supports cross-compiling specifications to Agda:

- Agda
- Rocq (on a branch soon to be merged)

ITP backend challenges

While cross-compiling to ITPs the main challenge we run into is maintainability - networks get retrained and the ITP model is will not be the canonical representation.

Current solution: Don't encode the network in the ITP, but implement "proof caching".

- Before starting verification, Vehicle hashes the network and stores the result.
- When checking the proof, Vehicle consults the proof cache and rehashes the network to check that it hasn't changed.
- Also drastically boosts performance in the ITP!

Trying Vehicle out

You can try it out:

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

Source code available at:

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

Open problems

Open problems in the Training backend

Remember the loss semantics for quantifiers:

$$[\![\texttt{forall}\,(\texttt{x} : \tau).\ e]\!]_l = [\![\bigwedge_{i=0}^{n}]\!]_l\,([\![e]\!]_l(x_i)) \qquad \text{where } x_i \sim \mathcal{D}([\![\tau]\!]_l)$$

Problem 1) - What does it mean for this to be sound? Clearly doesn't obey the previous definition, but maybe we could alter the definition:

$$\forall e.(\lim_{n\to\infty}[\![e]\!]_l = [\![\mathit{True}]\!]_l) \Rightarrow ([\![e]\!] = \mathit{True})$$

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...

Open problems in the verifier backend

Problem 2) Cyber-physical system specifications are often derived from the laws of physics which is (famously) non-linear. However, current solvers only support linear specifications.

Solution: Look at integrating solvers that can handle non-linear constraints. ("Provably Safe Neural Network Controllers via Differential Dynamic Logic", Teuber et al. to appear in NeurIPS2025)

Open problems in the Verifier backend

The generating error messages via type-systems works for checking if a specification is linear and first order, but abstract-interpretation verifiers solve a much more restricted set of problems...

e.g. α-β-Crown

- based on abstract interpretation $+$ MILP
- sound for first-order specifications that are in the chosen abstract domain (e.g. hypercube, zonotope, PRIMA, ...)

Problem 3) Is there a type-system that over-approximates membership to certain abstract domains, e.g. zonotopes? If not, how can we produce explanatory error messages for these solves?

Other problems

- **Problem 5)** Floating point semantics.
- **Problem 6)** Agent specifications involving probablistic guarantees.
- **Problem 7)** Increasing proof integrity via proof certificates.
- **Problem 8)** Quantised neural networks.
- etc.

See our paper for a more extensive list:

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

Conclusions

Conclusions

Vehicle allows users to write formal specifications for an AI agent and then facilitates training, verification and integration with upstream software designed for reasoning about the agent in the context of the environment.

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 of language and command line interface.

Collaborations

Moving forwards:

- There's a variety of interesting theoretical problems to be solved.
- We're keen to try it out on real-world problems!
- Eager to collaborate if people like any of these ideas or have others!