



Compositional Neural-Cyber-Physical System Verification in the Interactive Theorem Prover of Your Choice

MATTHEW L. DAGGITT, University of Western Australia, Australia

EKATERINA KOMENDANTSKAYA, Heriot-Watt University, UK and University of Southampton, UK

ALISTAIR SIRMAN, University of Southampton, UK

ALESSANDRO BRUNI, IT University of Copenhagen, Denmark

SAMUEL TEUBER, Karlsruhe Institute of Technology, Germany

JOSH SMART, University of Southampton, UK

GRANT PASSMORE, Imandra Inc., USA

Formal verification of neuro-symbolic cyber-physical systems, such as drones, medical devices and robots, is complicated. Neural components must be trained to be optimal with respect to the available data as well as the safety specifications, and then verified using specialised solvers. Symbolic models of the “cyber” and “physical” behaviour of the system must be constructed and verified in interactive theorem provers (ITPs), often requiring mature mathematical libraries to reason about the interplay of discrete and continuous dynamics, preferably obtaining infinite time-horizon guarantees. Finally, the results of the two already challenging verification tasks need to be integrated into a single proof. In this paper we present a compositional methodology for constructing such proofs. The *VEHICLE* framework provides a functional, domain-specific language for specifying, training, and verifying neural components. We extend *VEHICLE* to allow integration with any ITP with minimal effort, thereby bridging the gap between the neural and symbolic proofs. First, we describe how *VEHICLE*’s standard bidirectional type checker can be reused to transpile neural specifications into an intermediate representation targeting multiple theorem provers. Second, we integrate *VEHICLE* with Rocq, Isabelle/HOL, Agda and the industrial prover Imandra; and showcase a generic infinite time-horizon safety proof of a discrete cyber-physical system with a neural network controller in each ITP. Finally, to put the idea of *compositional neural-cyber-physical system verification* to the test, we use the Mathematical Components libraries in Rocq to verify infinite time-horizon safety of a medical device, modelled as a *continuous* cyber-physical system with a neural controller. To our knowledge, this is the first result of this kind in a general purpose ITP; and a result that was only feasible thanks to the compositionality provided by *VEHICLE*’s functional interface.

CCS Concepts: • **Software and its engineering** → **Syntax; Semantics; Domain specific languages**; • **Theory of computation** → **Logic and verification; Type theory; Automated reasoning; Program verification**.

Additional Key Words and Phrases: Neural Networks, Interactive Theorem Provers, Rocq, Agda, Isabelle/HOL

ACM Reference Format:

Matthew L. Daggitt, Ekaterina Komendantskaya, Alistair Sirman, Alessandro Bruni, Samuel Teuber, Josh Smart, and Grant Passmore. 2026. Compositional Neural-Cyber-Physical System Verification in the Interactive Theorem Prover of Your Choice. *Proc. ACM Program. Lang.* 10, ICFP, Article 282 (August 2026), 32 pages. <https://doi.org/10.1145/3828680>

Authors’ Contact Information: [Matthew L. Daggitt](mailto:matthew.daggitt@uwa.edu.au), University of Western Australia, Perth, Australia, matthew.daggitt@uwa.edu.au; [Ekaterina Komendantskaya](mailto:ekaterina.komendantskaya@hw.ac.uk), Heriot-Watt University, Edinburgh, UK and University of Southampton, Southampton, UK; [Alistair Sirman](mailto:alistair.sirman@南安普顿大学), University of Southampton, Southampton, UK; [Alessandro Bruni](mailto:alessandro.bruni@it.u Copenhagen), IT University of Copenhagen, Copenhagen, Denmark; [Samuel Teuber](mailto:samuel.teuber@kit.edu), Karlsruhe Institute of Technology, Karlsruhe, Germany; [Josh Smart](mailto:josh.smart@南安普顿大学), University of Southampton, Southampton, UK; [Grant Passmore](mailto:grant.passmore@imandra.com), Imandra Inc., Austin, Texas, USA.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2475-1421/2026/8-ART282

<https://doi.org/10.1145/3828680>

1 Introduction

Verification of cyber-physical systems (CPS) is a well-established research area that combines automated verification, reachability analysis, and differential equation solving [34]. There are several mature tools such as CORA [5], JuliaReach [13] and KeYmaera X [35], and their progress is evaluated annually in the ARCHCOMP competition [34]. In parallel, the Interactive Theorem Prover (ITP) community has also shown significant interest in verifying such systems, e.g. in Isabelle/HOL [14, 33, 80, 81], Rocq [14, 36, 55, 59], PVS [77] or ACL2 [15]. Generally, a CPS may be formalised and verified in a general-purpose ITP directly (examples in this paper follow this route), or reasoned about via domain-specific languages embedded in the ITP: e.g. *differentiable dynamic logic* (underlying KeYmaera X) has been implemented in PVS [77], Isabelle/HOL [14, 33] and Rocq [14].

A new generation of cyber-physical systems deploys neural network components [41, 49], leveraging the ability of neural networks to learn complex control strategies while remaining lightweight enough for real-time deployment. This has proven effective in a wide range of applications [34], from CERN cooling towers [49] to pilot advisory systems [41]. In this paper we refer to such systems as *neural-cyber-physical* systems (NCPS). Verifying the safety of these systems has been recognised as an important challenge in cyber-physical system verification [21, 48]. As illustrative example, consider the classic cyber-physical verification problem of Boyer et al. [15] where a car travels along a straight road while subject to cross-wind. The car's sensors provide noisy measurements of its environment, and in the original paper a symbolic controller is used to update its steering and velocity. The original safety property proven by Boyer et al. [15] states:

THEOREM 1.1. *Given that the wind can shift by no more than 1 per unit time, and the sensor is never off by more than 0.25 units, the car will never leave the road.*

In the CPS literature, this type of result is called an *infinite time-horizon* guarantee, as it ensures system safety for the entire execution of the system, rather than for a bounded time period.

Replacing the symbolic controller with a neural network transforms this example into a neural-cyber-physical-system where the car's software invokes the neural network at fixed time intervals to generate driving actions. While the safety property remains the same, we cannot prove it without also reasoning about the neural network's behavior.

Let us decompose the problem. An NCPS can be represented as $\mathcal{W}(\mathcal{S}(\mathcal{N}))$ where $\mathcal{W}$ is a model of the world the system operates in (i.e. the *physical*), $\mathcal{S}$ is a model of the conventional software components (i.e. the *cyber*), and $\mathcal{N}$ is the neural network controller (i.e. the *neural*). The verification goal is then to establish a safety property $\Phi(\mathcal{W}(\mathcal{S}(\mathcal{N})))$, for example that the car always remains within the road boundaries. However, while the cyber-physical components of the system, $\mathcal{W}$ and $\mathcal{S}$, can be represented and reasoned about *symbolically*, the neural component, $\mathcal{N}$, is a *sub-symbolic* model, i.e. it represents knowledge as numerical values rather than as explicit, human-readable symbols and hence admits no semantically meaningful decomposition. For example, the world model $\mathcal{W}$ and symbolic software $\mathcal{S}$ may contain the usual definitions of distance and velocity as functions of acceleration and time. In contrast, when $\mathcal{N}$ processes data that monitors distance, velocity and acceleration, its internal representation of this data is opaque and devoid of interpretable symbolic meaning, preventing us from decomposing $\mathcal{N}$ to reason about it symbolically and interactively.

The challenge is to develop methods that can combine proofs of safety of the symbolic cyber-physical system with proofs about the properties of its neural components.

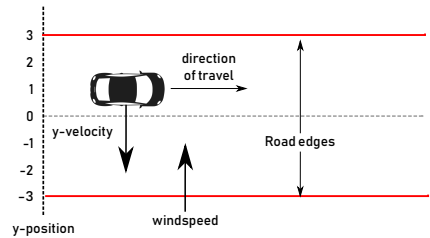


Fig. 1. Our running example of NCPS: A model of an autonomous car compensating for crosswinds.

1.1 Existing Automated Approaches

Motivated by this problem, a variety of techniques have been developed to prove the correctness of the neural component, $\mathcal{N}$, in isolation [3]. Linear programming-based neural network verification tools [9, 42, 47, 78] encode the neural network and desired property as mostly linear constraints, and apply linear constraint solvers that leverage sophisticated branch-and-bound techniques to handle the non-linearities. Abstract interpretation-based tools [64, 83] trade completeness for scalability to larger neural networks. Recent variants combine abstract interpretation with gradient descent [79] and Lagrangian optimisation [75] to recover completeness via branch-and-bound. While most tools limit themselves to verifying reachable (interval) bounds of a neural network or specifications given as linear constraints, some recent tools support nonlinear/polynomial constraints [62, 71]. We will refer to such tools as *neural solvers* in this paper.

These automated techniques have been extended to analyse a neural network $\mathcal{N}$ and its physical environment $\mathcal{W}(S(\cdot))$ in combination using a coupled reachability analysis, thereby obtaining safety guarantees about the entire system automatically [5, 13, 37, 39, 76]. They bypass the symbolic/sub-symbolic gap by implementing a combined reachability analysis in the same language: e.g. MATLAB in case of CORA [5], or C++ in case of PolarExpress [37, 76]. However, this convenience comes at a cost: rather than establishing invariants and infinite-time horizon safety properties, their guarantees are usually confined to *bounded* time horizons; moreover, performance degrades rapidly as system complexity or horizons grow [43]. This restricts the range of neural-cyber-physical systems that can be verified automatically. For example, these approaches would not help to prove Theorem 1.1.

1.2 The Compositional Approach

An alternative approach that overcomes these limitations takes inspiration from the decomposition $\mathcal{W}(S(\mathcal{N}))$. Starting from the desired system-level safety property Φ , one first derives a specification ψ of the neural network such that the proof of $\Phi(\mathcal{W}(S(\mathcal{N})))$ can be decomposed as follows:

$$\psi(\mathcal{N}) \quad \wedge \quad \forall f. \psi(f) \Rightarrow \Phi(\mathcal{W}(S(f))) \quad (1)$$

As shown in Figure 2, this separates reasoning about the *symbolic* cyber-physical system $\mathcal{W}(S(\cdot))$ from reasoning about the *sub-symbolic* neural component $\mathcal{N}$. This enables the use of existing ITPs to model and reason about complex symbolic dynamics (e.g. cyber-physical, probabilistic, perturbed, or partially observable systems) and obtain stronger guarantees about the full system (e.g. infinite-time horizon safety). The exact fragment of logic that ψ belongs to is unimportant as long as there is a specialised neural solver that supports it.

Although the compositional approach is appealing due to its potential for stronger guarantees, the only attempt to verify neural-cyber-physical systems in this way that we are aware of is Teuber *et al.* [70, 71]. However, their work did not integrate the proof of $\psi(\mathcal{N})$, obtained from a specialised neural solver (NCubeV [71]), and the proof of $\forall f. \psi(f) \Rightarrow \Phi(\mathcal{W}(S(f)))$, formalised in KeYmaera X [35]. Therefore, despite achieving infinite-time horizon guarantees, their results rely on the manual pen-and-paper composition of the proofs and lacks mechanised guarantees about alignment across the symbolic/sub-symbolic gap. We thus refine our thesis as follows:

*The challenge is to develop **compositional** methods for proving the safety of neural-cyber-physical systems.*

This is where functional programming offers a compelling solution. Functional languages provide a common framework in which both neural network interfaces and mathematical specifications (ψ) can be expressed with precise semantics. At the same time, they align naturally with the functional foundations of interactive theorem provers and can be translated readily into the declarative formats

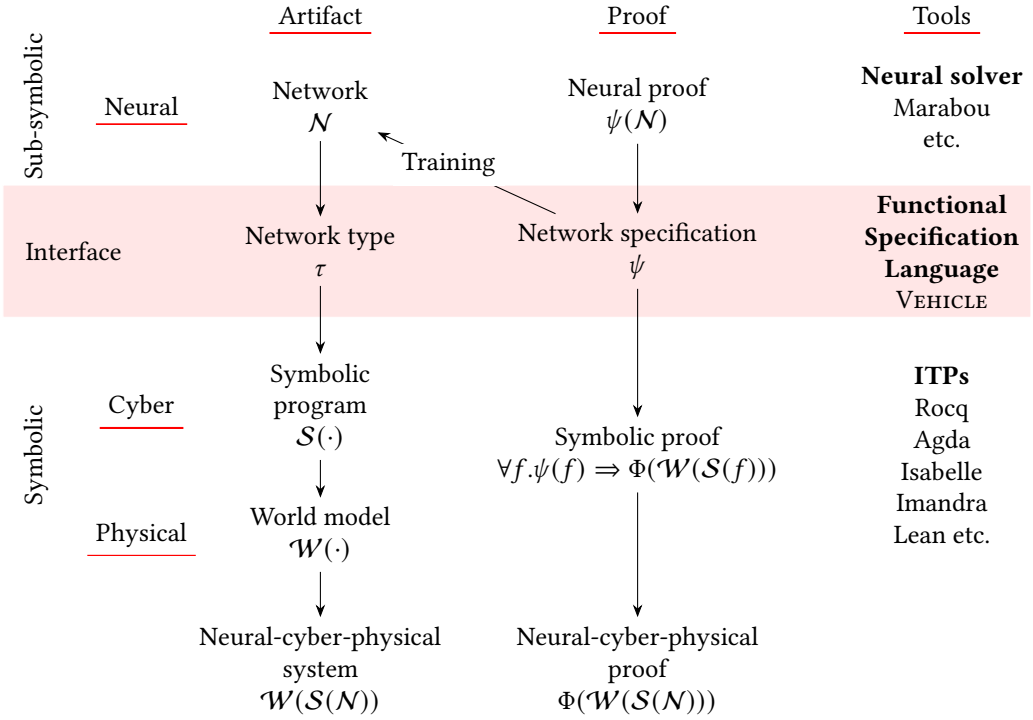


Fig. 2. The architecture of a compositional proof for a neural-cyber-physical system, with specifications in the VEHICLE DSL, in red, acting as an interface between existing neural solvers and existing ITPs.

expected by neural network solvers. As a result, functional programming provides an effective foundation for integrating symbolic and sub-symbolic proofs. Based on these observations, Daggett et al. [26] proposed VEHICLE, a functional domain specific language (DSL), for writing down the types and high-level specifications for neural networks. As shown in Figure 2, VEHICLE serves as the interface between the tools for verifying neural networks that operate in the sub-symbolic world and ITPs for reasoning about the larger symbolic system. Concretely, Daggett et al. [26] provided a high-level, non-technical description of how VEHICLE specifications are compiled to (i) loss functions for training the neural network, (ii) queries for the specialised neural solver Marabou [78] to verifying the resulting network, and (iii) Agda code for reasoning about the larger cyber-physical system.

1.3 Contributions

In this paper we make the following contributions:

- (1) **The first description of the internal implementation of the VEHICLE core language.** The three ecosystems targeted by VEHICLE – machine learning frameworks, neural network verifiers and ITPs – all have very different expressiveness and capabilities. Generating code for each of them requires unique analyses of the semantics of the specification ψ . In Section 3 we provide a concrete description of the architecture of the VEHICLE type-checker that allows us to reuse it to perform the backend-specific analyses in a modular fashion.¹

¹Although implemented, we do not provide a detailed description of the implementation of the learning and verification backends in this paper.

- (2) **A novel ITP intermediate language.** In the original proposal [26], `VEHICLE` only supported exporting neural network specifications to Agda. To address this limitation, in Section 4 we describe how the type-checker described in Section 3 can be used to compile the core `VEHICLE` language to a novel intermediate language used by the ITP backend that facilitates exporting the specification to any ITP with minimal additional code. We believe this technique is applicable to other domains which require translating Boolean specifications into ITPs.
- (3) **Integration with 4 mainstream ITPs.** In Section 5, we describe implementing support for exporting `VEHICLE` specifications to four mainstream general-purpose ITPs: Agda [53], Rocq/MathComp [2, 20, 72], Isabelle/HOL [52] and Imandra [56]. Together these systems cover both dependently/non-dependently typed systems and academic/industrial systems, demonstrating that our approach is applicable across the diverse ecosystem of ITPs. We describe the prover-specific design choices required in each case and compare the strengths and limitations of these systems for expressing neural-cyber-physical specifications. In particular, as an original contribution, we present a new Tensor library for MathComp [20] that is required to model neural network specifications. We prove infinite-horizon safety (Theorem 1.1) of the discrete NCPS in Figure 1 in each of the four provers.
- (4) **Infinite time-horizon safety of a continuous NCPS in a general-purpose ITP.** To demonstrate the advantages of our compositional approach, we present what we believe to be the first case study that verifies infinite-horizon safety of a continuous NCPS, a medical device, in a general-purpose ITP (Rocq + MathComp Analysis[2]). The techniques used in the case study can be generalised to any neural-cyber-physical system in which the property Φ guarantees the behavior known as *exponential decay* [63]. We show how recently released MathComp Analysis libraries (e.g. `derive.v` for multivariate derivatives and differentiation, `classical/filter.v` for continuity) can be used to prove properties of NCPS described by ordinary differential equations.

Together, these contributions provide a concrete demonstration of how functional programming can be used to bridge the symbolic/sub-symbolic gap in NCPS verification in a fully compositional manner and obtain new infinite-horizon safety guarantees that were not possible before. In particular, this work opens the way for existing [14, 33, 36, 55, 59, 80, 81] and future CPS formalisations in ITPs to integrate with neural network verification and training with minimal cost. We hope that this will significantly increase the attractiveness of general-purpose ITPs as a mainstream tool for the verification of neural-cyber-physical systems.

2 Recap: `VEHICLE` Surface Language

We begin by providing a short Agda formalisation of the cyber-physical components of the car example shown in Figure 1 and a brief recap of the `VEHICLE` surface language.

2.1 Wind Controller Example

In order to prove the safety of the NCPS introduced in Section 1, we start by defining a model in Agda of the cyber-physical components $\mathcal{W}(\mathcal{S}(\cdot))$ of the system as described by Boyer et al. [15]. The `State` datatype represents the state of the world, and the noisy sensor reading of its position obtained by the car. The `Observation` datatype represents updates to the world provided by an oracle, and finally the NCPS controller acts upon the sensor reading from the current and previous state, and instructs the car to change its velocity.

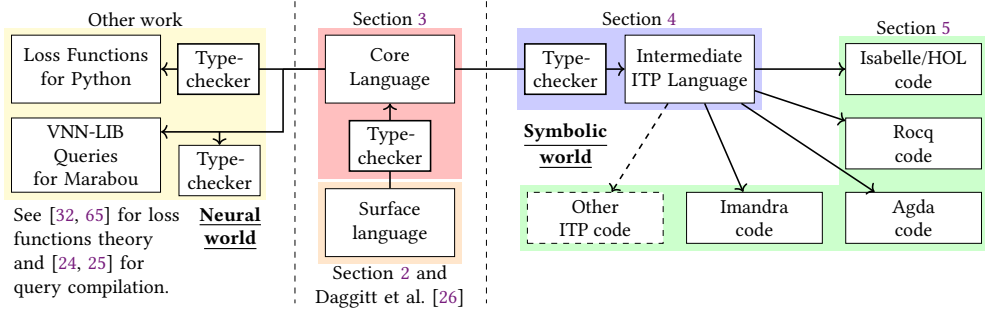


Fig. 3. The internal architecture of VEHICLE and the structure of this paper.

```

record State : Set
  where field
    windSpeed : ℝ
    position   : ℝ
    velocity   : ℝ
    sensor     : ℝ

record Observation : Set
  where field
    windShift : ℝ
    sensorError : ℝ

controller : Tensor ℝ [2] → ℝ
controller = ... -- network

```

Although the dynamics of the system is better modelled continuously (e.g. using ODEs), for simplicity we will model the cyber and physical components discretely using the following functions:

```

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 this model of $\mathcal{W}(S(\cdot))$, we can formally state the system property Φ described by Theorem 1.1.

```

OnRoad : State → Set
OnRoad s = | s.position | ≤ 3

ValidObservation : Observation → Set
ValidObservation o = | o.sensorError | ≤ 0.25 ∧
  | o.windShift | ≤ 1

systemSafety : ∀ xs → All ValidObservation xs → OnRoad (finalState xs)
systemSafety = ... -- symbolic proof

```

See the supplementary material of [26] for the full inductive symbolic proof of this infinite time-horizon guarantee. For the purposes of this paper, the crucial step is that the symbolic proof requires the neural component of the system controller to satisfy the following property:

```

safe : ∀ x → | x ! 0 | ≤ 3.25 ∧ | x ! 1 | ≤ 3.25
  → | controller x + 2 (x ! 0) - x ! 1 | < 1.25

```

This predicate refers *only* to the neural network controller, and thus is our neural specification ψ .

2.2 VEHICLE Surface Language

As shown in Figure 3, the core purpose of VEHICLE is to provide an external language in which users can write the type of the neural network τ and the neural specification ψ , use it to train and

verify a network, and then export τ and ψ to the ITP of their choice. The syntax for the VEHICLE surface language is shown in Figure 4 and the VEHICLE specification ψ for the car example can be written as shown in Listing 1.

$$\begin{aligned} \langle spec \rangle &\ni s = [d] \\ \langle decl \rangle &\ni d = v : t; v = e \mid @network\ v : t \mid @property\ v = e \mid @parameter\ v : t \\ \langle type \rangle &\ni t = t \rightarrow t \mid Bool \mid Real \mid Tensor\ t\ [n] \mid Index\ n \\ \langle expr \rangle &\ni e = e\ e \mid v \mid not\ e \mid e\ and\ e \mid e \Rightarrow e \mid forall\ v.\ e \mid if\ e\ then\ e\ else\ e \mid e \leq e \mid [e, \dots, e] \mid e!e \mid \\ &\quad b \in \mathbb{B} \mid i \in \mathbb{N} \mid r \in \mathbb{R} \end{aligned}$$

Fig. 4. Grammar for surface language of the VEHICLE DSL. This grammar is a restricted fragment of both the grammar in the full implementation and that presented in Daggitt et al. [26], sufficient to illustrate the challenges of compilation discussed in this paper.

Listing 1 The VEHICLE specification for the car’s neural network controller.

```

type InputVector = Tensor Real [2];   sensor1 = 0;   sensor2 = 1
type OutputVector = Tensor Real [1];   velocity = 0

@network controller : InputVector -> OutputVector

safeInput : InputVector -> Bool
safeInput x = -3.25 <= x ! sensor1 <= 3.25 and -3.25 <= x ! sensor2 <= 3.25

safeOutput : InputVector -> Bool
safeOutput x = -1.25 < controller x ! velocity + 2*(x ! sensor1) - (x ! sensor2) < 1.25

@property safe = forall x . safeInput x => safeOutput x

```

The property `safe` in the VEHICLE specification is semantically identical to the Agda lemma `safe` in Section 2.1. Given this specification VEHICLE can be used to train a neural network, via PyTorch, and then verify that the resulting file “model.onnx” satisfies the specification, via the neural solver Marabou [78]. Once this has been achieved, VEHICLE aims to generate Agda interface code for the network declaration and specification of the form:

```

controller : Tensor ℝ [2] → Tensor ℝ [1] | safe : forall x. SafeInput x → SafeOutput x
controller = callNetwork "model.onnx" | safe = checkVehicleProperty "spec.vclp"

```

where `callNetwork` is a macro that invokes the network and `checkVehicleProperty` is a macro that calls back to the VEHICLE compiler to check the status of the verification result. In this case, the VEHICLE compiler consults a cache that contains the location and a hash of the neural network model file used during verification, which it then uses to check the integrity of the proof avoiding reperforming verification of the network and disrupting the interactivity of the ITP.

VEHICLE goes against the prevailing trend in the verification community, where the vast majority of work integrating automated solvers into verification workflows in recent years has focused on implementing functional specification languages directly in the host language. Specifications are then compiled down to queries either by native macros or by the compiler itself. Examples are SMT or ASP solvers working underneath the hood in Lean [58], Isabelle/HOL [44], SMT-Coq [10],

F* [66], Liquid Haskell [74], Dafny [45] etc. The obvious question is why not follow this path and implement `VEHICLE` itself as an embedded compiler in the ITP, thereby allowing us to represent the neural network N and specification ψ in a single trusted logical framework? We argue that unfortunately this is infeasible for many reasons:

- (1) Even representing realistic neural networks in an ITP appears impractical. For example, the PyTorch [6] format has 3500 different operators and a trained network may have millions of parameters. Prior attempts at representing and verifying a concrete trained neural network in Rocq [8], Isabelle/HOL [16], and Imandra [30] have all reported issues with scalability of type checking and verification, and only support 3 operators.
- (2) Training the network to obey the specification ψ requires interfacing with state-of-the-art machine learning libraries (e.g. PyTorch, JAX, Tensorflow). We are unaware of any ITP that currently has support for this.
- (3) If the network's input tensors are large, even compiling ψ down to neural network queries may be prohibitively computationally expensive to perform in an ITP [25].
- (4) In NCPS systems, the final verified neural network often needs to be deployed on custom hardware such as ASICs or FPGAs. Again, there is no support for this workflow in any ITP that we are aware of.
- (5) Even if the above limitations were overcome, implementing the training and verification infrastructure in a single ITP would prevent it from being reused in other ITPs.
- (6) We hope that machine learning practitioners will be interested in using the training and verification backends of `VEHICLE` even if they do not wish to export their results to an ITP. Unfortunately, we believe they are unlikely to do so if they are required to work in an ITP instead of with the Python bindings provided by `VEHICLE`.

Therefore we argue that `VEHICLE`'s approach provides the most practical architecture for NCPS verification given the above limitations of current ITP technology. However, this approach does introduce two unavoidable trade-offs. First, it enlarges the trusted computing base of the ITP to include the `Vehicle` compiler, and therefore there is a possibility of introducing a semantic mismatch between `VEHICLE` and the target ITP. While this concern is valid, the `VEHICLE` language is deliberately minimal compared to a full ITP logic and, as described in Section 3, and specifications are monomorphised prior to export to further simplify its semantics. We therefore argue that the risk of semantic divergence in practice is low. Second, our approach treats the neural network as an abstract component and does not permit reasoning about its internal structure within the ITP. However, we are unaware of practical NCPS verification scenarios that require such internal reasoning. Moreover, if a NCPS verification task depends on semantically meaningful internal structure, we hypothesise that the network itself is decomposable into explicitly specified sub-components.

3 Implementation of `VEHICLE` Core Language

This section contains the first technical description of the Haskell implementation of the core `VEHICLE` language and its type-checker. One key challenge is that the three backends of the `VEHICLE` compiler, shown in Figure 3, each require unique semantic analyses of the neural specification ψ :

- (1) to compile the specification to a loss function, the training backend must determine which parts are differentiable.
- (2) to provide actionable error messages to the user, the solver backend must determine which parts of the specification are non-linear or contain alternating quantifiers; see [24].
- (3) to compile the specification to code for dependently typed ITPs, the ITP backend must determine which Boolean expressions in the specification are decidable internally in the ITP and which are not; see Section 4.

In `VEHICLE`, all of these analyses are implemented as auxiliary type-checking passes over the specification. With this in mind, the core Haskell AST datatypes in `VEHICLE` are defined as follows:

```
data Spec builtin = Spec [Decl builtin]

data Decl builtin
= Function Name (Expr builtin) (Expr builtin)
| Network Name (Expr builtin)
| Parameter Name (Expr builtin)
| Property Name (Expr builtin)

data Expr builtin
= Type
| Pi Name (Expr builtin) (Expr builtin)
| Lam Name (Expr builtin) (Expr builtin)
| App (Expr builtin) (Expr builtin)
| Var Ix
| Builtin builtin
```

By instantiating the `builtin` type parameter with different datatypes, we allow the language and type-system to be extended or restricted as appropriate in each backend. Note that we do not require the full power of extensible datatypes described in “Data types à la carte” [67] as we only need to extend the AST statically in a finite set of known ways.

Whilst parsing the specification, the AST is instantiated with the `StandardBuiltin` datatype:

```
data StandardBuiltin
= Nat -- Type
| NLit Int -- Nat
| List -- Type -> Type
| Nil -- List t
| Cons -- t -> List t -> List t
| Real -- Type
| Tensor -- Type -> List Nat -> Type
| TLit Tensor -- Tensor Real (size t)
| Forall -- (Tensor Real ds -> Bool) -> Bool
| Stack -- Tensor Real ds -> ... -> Tensor Real ds -> Tensor Real (cons d ds)
| Lookup -- Tensor Real (cons d ds) -> Index Real d -> Tensor Real ds
| Leq -- Tensor Real ds -> Tensor Real ds -> Bool

| Bool -- Type
| BLit Bool -- Bool
| Not -- Bool -> Bool
| And -- Bool -> Bool -> Bool
| Index -- Nat -> Type
| If -- Bool -> t -> t -> t
```

Note that real literals are implemented internally as zero-dimensional tensors.

We define the following Haskell type-class to express that an abstract `builtin` type can be type-checked:

```
class Typable builtin where
  convertBuiltin :: StandardBuiltin -> Expr builtin
  typeBuiltin :: builtin -> Expr builtin
```

where `convertBuiltin` is a function for mapping `StandardBuiltins` to the current `builtin` and `typeBuiltin` returns the type for each `builtin`. An instance of `Typable` for `StandardBuiltin` can be defined as follows:

```
instance Typable StandardBuiltin where
  convertBuiltin :: StandardBuiltin -> Expr StandardBuiltin
  convertBuiltin b = Builtin b

  typeBuiltin :: StandardBuiltin -> Expr StandardBuiltin
  typeBuiltin b = case b of
    ... -- See comments on StandardBuiltin datatype declaration above
```

We then define a function for type-checking declarations:

```
checkDecl :: (TCM m, Typable builtin) =>
  Decl builtin -> m (Decl builtin, [Constraint builtin])
checkDecl = ... -- standard bidirectional type-checker
```

$$\begin{array}{c}
\frac{\Gamma \vdash t : \text{Type} \quad \Gamma \vdash e : t}{\Gamma \vdash \text{fun } v : t = e} \text{ (FUNCTION)} \quad \frac{\Gamma \vdash e : \text{Bool}}{\Gamma \vdash \text{property } v = e} \text{ (PROPERTY)} \\
\\
\frac{t = \text{Tensor Real } ds_1 \rightarrow \text{Tensor Real } ds_2}{\Gamma \vdash \text{network } v : t} \text{ (NETWORK)} \quad \frac{\Gamma \vdash t : \text{Type}}{\Gamma \vdash \text{parameter } v : t} \text{ (PARAMETER)} \\
\\
\frac{\Gamma \vdash t_1 : \text{Type} \quad \Gamma, v \rightarrow t \vdash t_2 : \text{Type}}{\Gamma \vdash \pi(v : t_1).t_2 : \text{Type}} \text{ (PI)} \quad \frac{\Gamma \vdash t_1 : \text{Type} \quad \Gamma, v \rightarrow t_1 \vdash e : t_2}{\Gamma \vdash \lambda(v : t_1).e : \pi(v : t_1).t_2} \text{ (LAM)} \\
\\
\frac{\Gamma \vdash e_1 : \pi(v : t_1).t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2[v/t_1]} \text{ (APP)} \quad \frac{\Gamma[v] = t}{\Gamma \vdash v : t} \text{ (VAR)}
\end{array}$$

Fig. 5. The standard dependent-type system for the core VEHICLE lambda calculus.

This implements the type-system shown in Figure 5, where **TCM** m is a standard type-checking monad. We omit describing the details of the implementation, as it is a standard modern bidirectional type-checker for dependent-types (see Löh et al. [46] for details), where unknown types are represented using meta-variables and expressions are either checked against an expected type or a new type is synthesised. It also supports implicit and instance arguments, and the bidirectional type-checking pass generates a list of unification and instance/type-class constraints that need to be resolved by dedicated unification and instance solvers.

The full procedure for type-checking a declaration in VEHICLE is then defined as follows:

```

1 typeDecl :: (TCM m, TypableBuiltin builtin) => Decl StandardBuiltin -> Decl builtin
2 typeDecl decl = do
3   convertedDecl <- traverse convertBuiltin decl
4   (checkedDecl, constraints) <- checkDecl convertedDecl
5   (solutions, unsolvedConstraints) <- solveConstraints constraints
6   substDecl <- substitute solutions checkedDecl
7   generalisedDecl <- generaliseOver unsolvedConstraints substDecl
8   return generalisedDecl

```

First, the declaration is converted to use the current set of builtins before being bidirectionally type-checked, which generates a set of constraints and meta-variables to solve. After attempting to solve the constraints, a standard constraint generalisation procedure is applied to pi-abstract over unsolved meta-variables [40]. We present a concrete example of these steps in Section 4.

The full procedure for type-checking a specification is as follows:

```

1 typeSpec :: (TCM m, TypableBuiltin builtin) => Spec StandardBuiltin -> Spec builtin
2 typeSpec (Spec decls) = do
3   typedSpec <- traverseWhileUpdatingCtx typeDecl decls
4   monomorphisedSpec <- monomorphise typedSpec
5   return monomorphisedSpec

```

First, each declaration is type-checked in turn and then subsequently the whole specification undergoes a monomorphisation pass, which (i) specialises polymorphic definitions at each concrete type at which they are used and (ii) eliminates any type-class operations. The resulting program contains only concrete types and explicit operations, simplifying later compilation stages and facilitating translation to the training and verification backends where parametric polymorphism and type classes are unsupported.

The standard type-checking pass that every specification undergoes regardless of whether it is being used for training or verification or being exported to an ITP can then be defined simply as:

<code>@network</code>	<code>f : Real → Real</code>
<code>f : Real → Real</code>	<code>f = callNetwork "path/to/model.onnx"</code>
<code>neg : Bool → Bool</code>	<code>negBool : Bool → Bool</code>
<code>neg b = not b</code>	<code>negBool b = not b</code>
	<code>negProp : Set → Set</code>
	<code>negProp b = ¬ b</code>
<code>calc : Real → Real</code>	<code>calc : Real → Real</code>
<code>calc x = if neg (x <= 0) then 0 else 1</code>	<code>calc x = if negBool (x ≤^b 0) then 0 else 1</code>
<code>@property</code>	<code>safe : negProp (∀ x . calc x ≤ f x)</code>
<code>safe = neg (forall x . calc x <= f x)</code>	<code>safe = checkNetworkSpecification</code>
(a) Original VEHICLE specification	(b) Compiled Agda specification

Fig. 6. A VEHICLE specification where Boolean expressions must be specialised before compiling to the ITP.

```
typeCoreSpec :: (TCM m) => Spec StandardBuiltin -> Spec StandardBuiltin
typeCoreSpec = typeSpec
```

4 ITP Intermediate Language

When translating a specification to a dependently-typed ITP such as Agda, Rocq or Lean, for each Boolean subexpression we must decide whether to lift it to a proposition or whether it can remain at the Boolean level as there exists a decision procedure internal to the ITP capable of resolving it. To illustrate this, consider the example shown in Figure 6. In particular, notice that there can be no decision procedure for `forall x. calc x <= f x` in the definition of `safe` internally in the ITP, as there is no representation of `f` in the ITP and the ITP does not have access to the neural solver. Therefore it needs to be compiled to a proposition, and so the surrounding application of `neg` needs to be compiled to proposition-level negation. In contrast, the expression `neg (x <= 0)` in the definition of `calc` is in the body of an `if` and therefore `neg` at that location must be compiled to Boolean negation. Finally, in the unshown case where the user writes a quantifier inside an `if` condition then VEHICLE should be able to error gracefully rather than generating invalid ITP code.

One solution would be to make the distinction between booleans and propositions explicit in VEHICLE’s surface language. However we want to avoid this, as the distinction may be unfamiliar to machine learning users. Instead, this section describes a compiler pass capable of deciding if a given Boolean expression in a surface specification language must be resolved to a Boolean or Proposition in the extracted ITP code. In particular, we show how we can reuse the standard bidirectional type-checker and design a method for automatic generalisation to do so. To the best of our knowledge, a compiler pass implemented here is the first of its kind in the functional programming literature.

4.1 Implementation

To do so, we define a new datatype that extends `StandardBuiltin` with the corresponding type-level counter-parts of the supported Boolean operations.

```
data DecidabilityBuiltin
  = Standard StandardBuiltin
  | TrueType   -- Type
  | FalseType  -- Type
```

```

| AndType    -- Type -> Type -> Type
| ImpliesType -- Type -> Type -> Type
| LeqType    -- Tensor Real ds -> Tensor Real ds -> Type

```

The type **Spec DecidabilityBuiltin** will become the intermediate ITP language shown in Figure 3. The question is how we can define an instance of **Typable DecidabilityBuiltin** that allows us to reuse **typeSpec** to move from the **VEHICLE** specification in Figure 6a to a representation closer to the desired Agda code in Figure 6b.

The key trick is to define a new type-class *internally* in **VEHICLE** that has fields for the operations that can exist at either the type-level or the Boolean-level. There is deliberately no surface syntax for users to write instances in **VEHICLE**, so we write it Haskell-style as follows:

```

class Booleans where
  BoolTC    : Type
  TrueTC    : BoolTC
  FalseTC   : BoolTC
  AndTC     : BoolTC -> BoolTC -> BoolTC
  ImpliesTC : BoolTC -> BoolTC -> BoolTC
  LeqTC     : Tensor Real ds -> Tensor Real ds -> BoolTC

```

and we then define the two instances of this type-class as:

```

instance Booleans where
  BoolTC    = Bool
  TrueTC    = True
  FalseTC   = False
  AndTC     = And
  ImpliesTC = Or
  LeqTC     = Leq

instance Booleans where
  BoolTC    = Type
  TrueTC    = TrueType
  FalseTC   = FalseType
  AndTC     = AndType
  ImpliesTC = ImpliesType
  LeqTC     = LeqType

```

Note that these instances have no type parameters and therefore are overlapping, which is disallowed by default in languages such as Haskell and Agda (see Morris and Jones [51] for problems that can arise). However, as users cannot define instances directly in the **VEHICLE** surface language, we can support overlapping instances in the **VEHICLE** compiler without degrading the user experience.

Next, we define a conversion function from **StandardBuiltin** to **DecidabilityBuiltin** that uses the above definition to replace Boolean operations with their overloaded type-class equivalent:

```

convertToDecidabilityBuiltin :: StandardBuiltin => Expr DecidabilityBuiltin
convertToDecidabilityBuiltin b = case b of
  Bool      -> BoolTC
  BLit True -> TrueTC
  BLit False -> FalseTC
  And       -> AndTC
  Implies   -> ImpliesTC
  Leq       -> LeqTC
  _         -> Builtin $ Standard b

```

and define the types for **DecidabilityBuiltin** as follows:

```

typeDecidabilityBuiltin :: DecidabilityBuiltin -> Expr DecidabilityBuiltin
typeDecidabilityBuiltin = \case
  StandardBuiltin Forall -> ... -- (t -> Type) -> Type
  StandardBuiltin b      -> typeBuiltin b
  _                      -> ... -- See comments on declaration of `DecidabilityBuiltin'

```

where we overwrite the types of **Forall** to ensure it is always compiled to the type-level.

Given these two functions, the **Typable** instance for **DecidabilityBuiltin** can be defined as:

```

data Typable DecidabilityBuiltin

```

```

convertBuiltin = convertToDecidabilityBuiltin
typeBuiltin    = typeDecidabilityBuiltin

```

This **Typable** instance allow the existing instance constraint solver in the type-checker to determine the correct instance of **Booleans** to use at each location in the specification, and therefore the translation of the core language to the intermediate representation suitable for ITPs can be implemented simply by delegating to **typeSpec** which was defined in Section 3:

```

toIntermediateITPLang :: TCM m => Spec StandardBuiltin -> m (Spec DecidabilityBuiltin)
toIntermediateITPLang = typeSpec

```

4.2 Worked Example

To illustrate this claim, we step through the operation of **toIntermediateITPLang** in the **VEHICLE** specification in Figure 6a. Note that for readability, we use the surface **VEHICLE** syntax rather than the core syntax. First, we consider the operation of **typeDecl** on the **VEHICLE** function **neg**. Initially, the following code is obtained after the builtins have been converted in Line 3 of **typeDecl**:

```

neg : BoolTC -> BoolTC
neg b = NotTC b

```

At the end of constraint solving in Line 6 of **typeDecl** there is insufficient information to work out which implementation of **Booleans** to use and so we end up with the following:

```

neg : BoolTC {{?0}} -> BoolTC {{?0}}
neg {{?0}} b = NotTC {{?0}} b

```

where $\{\{ \cdot \} \}$ is the standard double braces notation for instance arguments, and meta-variable $?0$ represents the unsolved **Booleans** instance. The application of constraint generalisation in Line 7 of **typeDecl** will cause this unsolved meta-variable to be universally quantified and prepended to the type to obtain the final generalised version of the function that works over either at the level of **Booleans** or at the level of types:

```

neg : {{i : Booleans}} -> BoolTC {{i}} -> BoolTC {{i}}
neg {{i}} b = NotTC {{i}} b

```

However when processing the declaration of **calc**, the type-checker *will* be able to resolve which implementation to use, as according to the type system the condition of an “if” must be of type **Bool** and so the **neg** here must use the plain Boolean instance (denoted here by **BI**):

```

calc : Real -> Real
calc x = if neg {{BI}} (x >= 0) then 0 else 1

```

Similarly, the type-checker will be able to deduce that the call to **neg** must use the type instance (denoted here by **TI**) due to the type of **forall**:

```

safe : Type
safe = neg {{TI}} (forall x . calc x <= f x)

```

The monomorphisation pass in **checkSpec** will then specialise the newly polymorphic **neg** declaration into two separate definitions, giving us the following final code:

```

negBool : Bool -> Bool      calc : Real -> Real
negBool b = not b          calc x = if negBool (x >= 0) then 0 else 1

negType : Type -> Type     safe : Type
negType b = notType b      safe = negType (forall x . calc x <= f x)

```

This code is now a simple conversion of the builtins away from the final desired Agda code shown in Figure 6b (or indeed any other ITP). We now present a proof of soundness of this translation.

Table 1. Comparison of ITP backends. LOC = line of code. No* = introduced as part of this development.

Interactive Theorem Prover	Agda	Rocq	Isabelle/HOL	Imandra
Dependent types	Yes	Yes	No	No
Pre-existing tensor libraries	No*	No*	Yes	No*
ITP - tensor library LOC	192	939	783	1055
ITP - other LOC	204	219	178	684
Haskell LOC in <code>VEHICLE</code>	772	720	981	862
Proof cache integration	Yes	Yes	No	No

THEOREM 4.1 (SOUNDNESS OF ITP TRANSLATION). *Let s be a well-typed specification, then if `toIntermediateITPLang` terminates on s successfully yielding s' , then s' is also well-typed and $s = s'$ everywhere except for boolean expressions, where every boolean expression in s corresponds to a set of one or more semantically equivalent expressions in s' .*

PROOF. Assuming that the implementation of the bidirectional type-checker, constraint solvers and monomorphisation are sound, then we observe that the translation is the composition of three distinct semantic-preserving stages. First, the initial conversion pass overloads only the Boolean builtins via the `Booleans` type-class. Next, the success of the constraint solver implies that there exists a valid partition of the declaration's boolean logic into Boolean and propositional sorts. Lastly, monomorphisation guarantees that all polymorphic definitions (e.g., `neg` in Figure 6) are specialised, thereby ensuring that all references to the `Booleans` type-class are eliminated and resulting in a target term e' containing only concrete, disjoint `Bool` or `Type`-level operations from `DecidabilityBuiltin`. Thus, by construction, and, as s is already well-typed, if `toIntermediateITPLang` succeeds, e' is a well-typed instance of e where the ambiguity of boolean operations has been resolved into explicit decidable or propositional forms. $\square$

5 Integration with an ITP of Your Choice

Given the implementation pipeline described in Section 4, unlocking compositional verification of neural-cyber-physical systems in a new ITP X consists of the following two steps:

- (1) creating a `VEHICLE` companion library in X that contains missing definitions in X (e.g. tensors) and, optionally, a macro that calls back to the `VEHICLE` compiler when checking the proof;
- (2) adding a Haskell function `compile : Spec DecidabilityBuiltin -> String` to the `VEHICLE` compiler that translates `VEHICLE` specifications to code targeting the ITP X and its companion library.

We have performed these steps for 4 widely-used interactive theorem provers: Agda, Rocq, Isabelle/HOL and Imandra. Table 1 provides a comparison of each implementation. For ITPs that already contain a formalisation of tensors (i.e. Isabelle/HOL), the size of the ITP companion library is extremely small (178 LOC). Similarly, the implementation of compilation to the ITP within `VEHICLE` requires less than 1000 LOC in all systems. In contrast, `VEHICLE`'s integration with neural network training and verification ecosystems amounts to 60,000 LOC of Haskell and 4,000 LOC of Python. This underlines the value of integrating ITPs via `VEHICLE` as opposed to developing custom infrastructure for NCPS verification in each ITP individually.

We will now discuss the implementation in each of the 4 interactive theorem provers, comparing the suitability of each system and its libraries for cyber-physical verification.

5.1 Agda

As Agda is dependently typed, `VEHICLE` specifications can be compiled directly one-to-one to Agda code. Section 2 has already given an overview of the Agda code that is generated by the Agda backend. Therefore we will only briefly discuss the `VEHICLE`-Agda companion library. Agda was chosen as the first ITP we added to support for primarily due to familiarity with the system, rather than a belief that it is the best ITP for performing verification of cyber-physical systems. Indeed, it has proven severely lacking in library support, as not only does the Agda Standard Library [23] not contain a formalisation of ODEs or tensors, it does not even contain a standardised representation for real numbers. Therefore, the `VEHICLE`-Agda companion library contains a very basic formalisation of tensors, and we are forced to compile real numbers to Agda rationals. The latter is sound as long as the network does not contain any transcendental operations (e.g. `tanh`). The advantage of `VEHICLE` is that rather than investing serious effort in fixing these deficiencies in the Agda libraries, we can instead move to an ITP with better library support.

5.2 Rocq

Rocq is the second ITP we implemented support for in `VEHICLE`. We chose to target the MathComp [20] and MathComp Analysis [2] libraries for this backend, as they contain an extensive formalization of mathematics that goes well beyond what both the Rocq and Agda standard libraries provide. Targeting Rocq and MathComp Analysis allows us to leverage its formalization of real numbers and analytical arguments. As part of this paper's contribution, we complement Rocq/MathComp with a novel general implementation of tensors.²

5.2.1 MathComp Integration. The MathComp [20] and Analysis [2] libraries contain a comprehensive collection of Rocq formalization of mathematical structures. MathComp focuses on the constructive formalization of algebraic structures. In particular, we exploit the hierarchy of algebraic structures (i.e., monoids, semigroups, $\mathbb{Z}$ -modules, semirings and rings) to construct appropriate instances for our tensor library. As we will see later, we define tensors by wrapping matrices and appropriately translating indices, so most of these interface implementations are one-liners.

Many `VEHICLE` constructs map directly onto MathComp definitions. For example, `VEHICLE`'s `Index n` representing the finite set $\{m \in \mathbb{N} \mid m < n\}$, corresponds to MathComp ordinals (`'I_n`). Other `VEHICLE` constructs lifted to MathComp include lists with operations such as folding and mapping, ordering relations, equality, and boolean operations. Our translation allows users to leverage mature, well-tested formalizations and benefit from MathComp's extensive lemma libraries. We translate `VEHICLE` reals to the formalization of real numbers in MathComp Analysis, which are accessed via an interface and thus compatible with the constructive definition present in the Rocq standard library, which we exploit whenever we need to compute concrete inequalities.

5.2.2 Tensor Implementation. Tensors are fundamental to `VEHICLE`'s design: neural network declarations require tensor types for both inputs and outputs, and, as discussed in Section 3, internally all scalar values are represented as zero-dimensional tensors. However, MathComp lacks a tensor formalization, necessitating a custom implementation.

We designed the tensor library to satisfy two goals: (1) support the subset of tensor operations required for `VEHICLE` specifications, and (2) provide a general formalization suitable for eventual inclusion in MathComp itself. The key insight is to represent tensors as matrices from MathComp's existing `algebra/matrix.v` module [20]. A tensor $T_{j_1, \dots, j_m}^{i_1, \dots, i_n}$ with contravariant indices $i_1 \in I_1, \dots, i_n \in I_n$ and covariant indices $j_1 \in J_1, \dots, j_m \in J_m$ is encoded as a matrix

²Our tensor implementation has been merged into MathComp [17]

Listing 2 Rocq tensor definition: tensors as matrices.

```

Context {n m : nat} (u_ : nat ^ n) (d_ : nat ^ m) (K : Type) (R : pzRingType).
Variant tensor_of : Type := Tensor of 'M[K]_(\prod_(i < n) u_ i, \prod_(i < m) u_ j).
Notation "'T[' R ]_ ( u_ , d_ )" := (tensor u_ d_ R).
  
```

$M = (r_{ij})_{1 \leq i \leq M, 1 \leq j \leq N}$ where $M = |J_1| \cdot \dots \cdot |J_m|$ and $N = |I_1| \cdot \dots \cdot |I_n|$. In the general tensor notation, covariant indices represent basis vectors and transform inversely under coordinate changes, while contravariant indices represent coordinate displacements and transform directly (see e.g., [31]). The Rocq definition is shown in Listing 2, where nat^n represents a finite function from I_n to nat and $M[K]_n(m)$ is a matrix of size n times m . Note that this encoding supports both covariant and contravariant indices, though VEHICLE itself uses only contravariant tensors. This generality enhances the library’s potential utility beyond VEHICLE.

The matrix-based representation allows us to inherit algebraic properties from MathComp’s matrix library. For example, the Hadamard (pointwise) tensor product is defined as:

$$(T \odot U)_{j_1, \dots, j_m}^{i_1, \dots, i_n} = T_{j_1, \dots, j_m}^{i_1, \dots, i_n} \cdot U_{j_1, \dots, j_m}^{i_1, \dots, i_n}$$

In Rocq we define this as:

```

Definition hmult (t u : 'T[R]_(u_, d_)) :=
  @Tensor _ _ u_ d_ R (map2_mx *%R (\val t) (\val u)).
  
```

This definition uses the `map2t` function that applies an operator (e.g., ring multiplication `*%R`) pointwise.³ We then inherit all algebraic structure from the Hadamard matrix product, e.g., to instantiate the ring interface we prove distributivity:

```

Lemma hmultDl : left_distributive hmult +%R.
Proof. by move=> x y z; rewrite /hmult map2_mxDl. Qed.
[... ]
HB.instance Definition _ := GRing.Nmodule_isPzSemiRing.Build
  'T[R] hmultA hmult1t hmult1 hmultDl hmultDr hmult0t hmult0.
  
```

The resulting tensor library implements standard interfaces including equality, addition, subtraction, pointwise partial ordering, and the Hadamard product, all matching VEHICLE’s tensor semantics, as well as other definitions of general utility, including the tensor product and its bilinear properties.

5.2.3 Neural Network Representation and Axiomatization. Compiled VEHICLE specifications represent neural networks as abstract functions with axiomatized properties. When VEHICLE compiles a specification containing a `@network` declaration, the generated Rocq code includes:

- (1) An *opaque function definition* for the network, hiding its internal structure. The network weights and architecture are not exposed in the generated proof script.
- (2) *Axioms* asserting that the network satisfies the verified properties. These axioms are justified by the external verification step performed by the neural network verifier (e.g., Marabou).
- (3) *Type signatures* specifying the network’s input and output tensor dimensions.

These components look very similar to analogous components in Agda, exemplified in Section 2; further examples in Rocq are given in Section 6 and Appendices A.2 and B.

5.3 Isabelle/HOL

The third ITP we added support for was Isabelle/HOL [52], an important addition seeing that majority of ITP CPS formalisations were given in this prover [14, 33, 80, 81]. We begin this section

³The `@` symbol makes all parameters explicit, hence the need to specify the real type of both operands and the result, as well as the dimension arguments `us` and `ds`.

by describing the `VEHICLE` companion library and how we bridge `VEHICLE`'s and Isabelle/HOL's type system despite Isabelle/HOL's lack of dependent types. Subsequently, we describe the compilation of `VEHICLE` properties to Isabelle/HOL.

5.3.1 *VEHICLE-Isabelle/HOL Companion Library.* In contrast to Rocq MathComp, Isabelle/HOL already has a formalization of Tensors available, which is part of the AFP's *Deep_Learning* package [11]. Hence, we reuse this formalization for our companion library.

Type System. The tensor type of `VEHICLE` is mapped to the tensor type `'a tensor`. In contrast to Agda and Rocq, Isabelle/HOL does not support dependent types. Hence, while Isabelle/HOL's tensor type is parametric in its element type (`'a`) it does not support constraining tensors to a particular shape. Consequently, Isabelle/HOL's tensor addition implementation checks shape compatibility during evaluation and returns the type's default member undefined in case of incompatible shapes. Fortunately, `VEHICLE`'s type checker statically analyzes shape compatibility thus ensuring that no ill-shaped tensor operations appear in the generated assumptions.

Nonetheless, we must ensure that we preserve tensor shape information for the networks represented inside Isabelle/HOL. If a neural network's return type is only given as `'a tensor`, extracting a component may return undefined unless the tensor is of suitable shape which we would not know from the type alone. To this end, we leverage Isabelle/HOL's semantic subtyping infrastructure [52, Sec. 8.5.2] to preserve the typing information in Isabelle/HOL despite its lack of dependent types. This allows us to construct a type of all tensors of a particular shape. For example, we can define a real valued input vector of shape [2] as follows:

```
typedef InputTensor = "{ a :: R tensor. (dims a) = [2] }"
using dims_tensor_from_lookup by blast
```

Type Coercion. In the context of larger expressions, elements of custom subtypes like `InputTensor` inevitably have to interact with other tensor types. By default, this would lead to type errors. While this can be resolved by explicitly wrapping all variables in appropriate conversion functions, this would make expressions unwieldy.

Instead, we rely on *automatic coercion* functions which allow us to instruct Isabelle/HOL to automatically convert, e.g., from `InputTensor` to `R tensor`. Ideally, we would allow seamless bidirectional coercions between `InputTensor` and `R tensor`. However, Isabelle/HOL does not allow for cyclic coercions. To circumvent this, we define an (isomorphic) tensor type `'a FlexTensor`. We leverage this additional type to break cycles by ensuring that all functions take *inputs* of type `'a tensor`, and *outputs* of type `'a FlexTensor`. We then define automatic coercions from custom subtypes to `'a tensor` as well as coercions from `'a FlexTensor` to `'a tensor`. This enables arbitrary composition of different tensor types (with matching shapes guaranteed by `VEHICLE`).

Our companion library comprises the definition of `'a FlexTensor`, its automatic coercion functions, and wrapper functions for the tensor operations provided by *Deep_Learning* which satisfy the typing constraints outlined above.

5.3.2 *Compilation of VEHICLE specifications to Isabelle/HOL.* The high-level approach of `VEHICLE`'s Isabelle/HOL backend is similar to the strategy pursued by the Agda and Rocq backend, i.e., a given `VEHICLE` specification file is translated into an Isabelle/HOL theory by mapping `VEHICLE` expressions to Isabelle/HOL expressions. Unlike Agda, `VEHICLE`'s Isabelle/HOL backend currently has no support for a proof cache. We now discuss compilation of custom tensor types, neural network properties and function definitions, in turn.

Compilation of Custom Tensor Types. Tensor types with a specific shape, like `InputTensor`, are compiled to **typedef** declarations as discussed above. In addition to the definition itself, `VEHICLE`

also generates a proof of the non-emptiness of the generated type which is required by Isabelle/HOL. `VEHICLE` also defines and registers the automatic coercion from `'a FlexTensor` to `InputTensor` as well as from `InputTensor` to `'a tensor`. Finally, each custom type is equipped with a commonly needed simplification lemma (and its proof) enabling easy access to a tensor's shape and elements:

```
lemma InputTensor_tensor_rewrite0[simp]:
  assumes "prod_list shape = length elems"
  and "shape = (2 :: nat) # []"
  shows "(Rep_tensor (Rep_InputTensor (Abs_InputTensor (Abs_tensor (shape,elems))))))
    = (shape,elems)"
```

Networks and their properties. To modularize the reasoning about neural networks and their properties, we translate a given `VEHICLE` specification file into an Isabelle/HOL **locale**. A locale combines a set of fixed parameters with assumptions about these fixed parameters which can be used in subsequent definitions, lemmas and proofs. Locales can also be extended or combined with other locales in downstream theories to allow composition of different sets of assumptions. In our case, we define the neural networks as fixed parameters of a locale and formalize the properties of the network provided by `VEHICLE` as assumptions of the locale. By relying on Isabelle/HOL's locale infrastructure, we avoid introducing the neural network properties directly as axioms in the global scope (which would be unproven from the perspective of Isabelle/HOL's core). The resulting locale is of the following form:

```
locale WindCtrl =
  fixes controller :: " InputVector => OutputVector"
  assumes safe:"(\<forall> x. (safeInput controller x) --> (safeOutput controller x))"
  begin end
```

To prove system-level properties for a given neural-cyber-physical system, Isabelle/HOL users can import the `VEHICLE` generated file into their theory. Subsequently, they can leverage the fixed parameters and assumptions of the locale in their proofs. For example, we can define a scalar version of the DNN's execution within the locale's context as follows:

```
context WindCtrl
begin
  definition controller_scalar :: "real => real => real"
  where "controller_scalar p1 p2 = (
    lookup (Rep_OutputVector (controller
      (normalise controller (Abs_InputVector (tensor_from_vec [2] [p1, p2])))))
      [WindControllerSpec.velocity])"
  lemma finalState_safe: (* ... *) end
```

Function Definitions. Function definitions are constructed as regular Isabelle/HOL definitions. As seen in the example above, the definition of the neural network only exists relative to the locale `WindCtrl` while the definitions of `safeInput` and `safeOutput` happen outside the locale. Consequently, we extend all definitions by additional parameters for the referenced neural network (e.g. `safeInput` now has as first input the neural network controller). See A.3.

5.4 Imandra

Imandra is an industrial automated reasoning system whose modelling language, IML, is based on a pure subset of OCaml [56]. It provides support for infinite-precision reals [28], which we make use of in this paper. Imandra is used in production for formal verification in finance by firms such as Goldman Sachs, Broadridge and Citi. There is a history of neural network verification in Imandra: fully executable neural network representations in IML have been subjected to proofs and counterexample analysis [29, 30]. In addition, verification of autonomous (cyber-physical)

systems is part of Imandra's portfolio.⁴ Thus, combining the two themes bears promise. Imandra's logic is classical and all propositions are expressed as executable boolean-valued predicates, so `VEHICLE`'s decidability distinction maps straightforwardly. A key feature of Imandra's core logic is a semi-decision procedure that is complete for counterexamples in a precise sense, even in the presence of recursion, nonlinearity and higher-order functions [56].

5.4.1 Type System. Like Isabelle, Imandra is not dependently typed. As part of this work, we developed a tensor library for IML (1,739 LOC across 5 IML files) providing tensor operations with algebraic properties proven within Imandra. Tensors are represented as records with well-formedness maintained by smart constructors and validity predicates:

```
type 'a tensor = { dims : int list; vec : 'a list }
let tensor_from_vec (d : int list) (v : 'a list) : 'a tensor = { dims = d; vec = v }
```

5.4.2 Translation Strategy. Neural networks are declared as opaque IML functions using the `[@@opaque]` attribute, and externally verified properties become axioms (typically expressed using tensor arithmetic) about the opaque neural network. For example, the code below shows some of the components of the compiled wind controller specification:

```
let controller : real tensor -> real tensor = () [@@opaque]
(* ... *)
axiom safe x = safe_input x ==> safe_output x
```

The full compiler-generated Imandra file is shown in Appendix A.4. In future work, we envision importing `VEHICLE`'s externally found proofs alongside more extensive executable representations of the neural network controllers expressed in Imandra's logic, so as to take advantage of Imandra's verified neural network proof checker [29] which can import proofs from external neural network verifiers such as Marabou.

To scale to more complex examples of NCPS, we will need to upgrade some of the standard Imandra libraries. Imandra's real type is the full field of real numbers: input-language literals (such as `3.14 : real`) are written as rationals, but are interpreted as reals with reasoning taking place using the theory of real closed fields (and computational support for computing with these values taking place via the minimal real closed field, the real algebraic numbers). What is absent from the standard libraries is the more general apparatus of real analysis: transcendental functions such as `exp` and `ln`, limits, derivatives, and certified interval arithmetic over transcendental expressions.

Because Imandra is a computational logic in which every definition is executable, the natural path is to develop analysis computationally rather than classically. E.g., `exp` and `ln` would be defined as fast-converging Cauchy sequences of rationals equipped with a computable modulus of convergence; continuity and derivatives would be stated and proved in the computable-analysis style [12, 54]; and range constraints over reals would be discharged by a verified interval-arithmetic tactic built on those definitions. The resulting library would be genuinely different from Rocq's MathComp Analysis, as every theorem would carry a computation that can be evaluated on concrete inputs. Though nontrivial, this is mostly an engineering programme that is very much in the spirit of Imandra's design. We expect the Imandra community to take it on over time.

6 Case Study: Infinite-Horizon Safety of a Continuous Neural-Cyber-Physical System

Neural networks promise to give a viable alternative to traditional Bayesian estimation methods in the medical domain [57]. This case study concerns a medical device that administers an antibiotic Vancomycin, taking into account the patient's current drug concentration, temperature, white blood cell count, age, and weight, using simulated data for training. We are assuming that the

⁴<https://www.imandra.ai/autonomous-systems>

neural network was trained to administer an optimal dose given observations based on these five parameters and has a type $\mathbb{R}^5 \rightarrow \mathbb{R}$. As drug dosing is a safety-critical application, we also wish to obtain worst-case guarantees about the safety of dosing. Using the same methodology as in Section 2.1, we will first discuss the overall system safety, and then discuss the role of neural network verification in it.

This case serves to discuss the situation when the larger system verification may involve reasoning about the system dynamics explicitly: in particular, reasoning about non-linear functions and their derivatives. Such verification is best served by an ITP whose real-analysis libraries are mature enough to handle the elementary calculus directly and whose numerical-bound machinery can discharge the transcendental comparisons that arise. Our case study draws on MathComp-Analysis [2] for derivatives, continuity, and integration, and on CoqInterval [50] for verified real-valued inequalities. Beyond this experiment, the wider Rocq ecosystem already supports richer dynamic-systems verification through LaSalle's invariance principle for stability arguments [19] with demonstrated case studies (e.g., an inverted pendulum [60]) and recent analytic-ODE solutions [73]. The further additions needed to reach the state of the art in full continuous-dynamics verification are modest: principally, a general-purpose Picard–Lindelöf theorem and the resulting flow theory.

Table 2. Mathematical libraries used in case study. The Rocq proof script is approximately 1,200 lines and with roughly 860 apply/rewrite/exact tactic invocations drawing on about 200 distinct library identifiers.

Library / module	Approx. uses
MathComp	
ssralg, ssrnum, order	~100 lemmas
bigop, seq, tuple, ssrnat	~15 lemmas
MathComp Analysis	
Derivatives [2]	~25 lemmas
Topology, filters, limits	~20 lemmas
Real exponentials and logarithms	~20 lemmas
VEHICLE (tensor primitives)	~15 lemmas
CoqInterval [50]	10 interval tactic invocations
lra	25 invocations

Definition `derive (f : V -> W) a v := lim (`
`(fun h => h^-1 * ((f \o shift a) (h * v) - f a)) @ 0^').`

Notation `'''D_' v f c" := (derive f c v).`

Definition `derive1 V (f : R -> V) (a : R) := lim (`
`(fun h => h^-1 * (f (h + a) - f a)) @ 0^').`

Notation `"f ^ (^)" := (derive1 f).`

Definition `continuous_at (T U : nbhsType) (x : T)`
`(f : T -> U) := (f%function @ x --> f%function x).`

Notation `continuous f := (forall x, continuous_at x f).`

$$D_0 f(x) = \lim_{h \rightarrow 0} \frac{f(x + hv) - f(x)}{h}$$

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$

$$\lim_{a \rightarrow x} f(a) = f(x)$$

Fig. 7. MathComp Analysis definitions of multivariate derivative, differentiation and continuity and their mathematical counterparts.

Table 2 summarises the Rocq libraries the medical case study draws on, with approximate usage counts. The bulk of the proof relies on MathComp algebra/order and the MathComp Analysis derivative layer (including lemmas reasoning about maxima of monotone functions); bridging to

Stdlib's `reals` via `Rstruct/Rstruct_topology` is required to invoke `CoqInterval`, which in turn discharges the approximation arguments.

6.1 System Safety in Rocq

As shown in Figure 9, the state consists of the patient information. Furthermore, `n_doses` provides a function that returns the sequence of doses chosen by the neural network. This is because, unlike the Wind Controller example, the concentration at any point in time is dependent on all previous doses, and this information cannot be obtained from *only* the previous state. Note that `n_doses` uses the neural network called `network` to perform the computation. The network used here is in a reduced form, where all information except the concentration has been partially applied, making it of type $\mathbb{R} \rightarrow \mathbb{R}$. In line with the other example, it is this occurrence of the neural network that raises concerns about the overall system safety.

To model drug concentration after a single dose, we follow the standard exponential decay *pharmacokinetic equation* [68] shown in Figure 10. Pharmacokinetic equations are a system of ODEs that describe how the body processes doses of drugs. In our case, we focus on the *one-compartment* pharmacokinetic equations, which is a more simplistic model that considers the body as a single conglomeration of blood. More intricate models can be found in Talevi and Bellera [69]. The corresponding function `Concentration` takes as input the dose D and time t . As parameters, it has the volume of blood V_d and an absorption and elimination coefficients k_a and k_e that characterise, respectively, the rates of absorption and elimination of the drug in the patient's blood as seen in Figure 10.

To model drug concentration after a series of doses, we define the function `total_conc` in Figure 11 that takes as a parameter `Ds`, a tuple of all doses given to the patient at regular dosing intervals (`tt`) and takes the sum of all concentrations from all doses given a point in time. Note that, in the context of MathComp, an `n`-tuple `R` is a list of length `n` of type `R`:

<pre> Fixpoint n_doses (initial : R) (n : nat) : n.+1-tuple := match n with 0 => [:: network initial] n'.+1 => let Doses := n_doses initial n' in rcons Doses (network (total_conc D'.+1))%R) end. </pre>	<pre> Record state := State { C : R ; T : R ; wbc : R ; age : R ; weight : R }. </pre>
---	---

Fig. 9. The Rocq definition for the state and the iterative function (similar to `next_state` in subsection 2.1). Note that `network` is a reduced form that only takes the concentration as its argument.

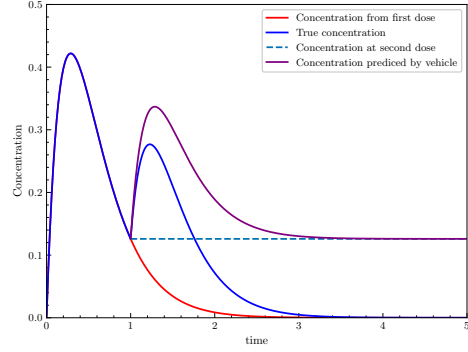


Fig. 8. A graph showing how the concentrations are interpreted by different parts of the verification process. The red line is equal to the blue line until 1, representing Figure 10 and Figure 11, respectively.

$$\text{Definition Concentration } (D \ t : \mathbb{R}) : \mathbb{R} := \left((D * K_a) / (V_d * (K_a - K_e)) * (\exp R ((-K_e) * t) - \exp R ((-K_a) * t)) \right) \quad \left| \quad C(D, t) = \frac{D \cdot k_a}{V_d \cdot (k_a - k_e)} \cdot (e^{-k_e \cdot t} - e^{-k_a \cdot t}) \right.$$

Fig. 10. Pharmacokinetic equation as a concentration function relying on the absorption and elimination coefficients k_a, k_e , blood volume V_d and dose D at time t . Adapted from Equation 15 in “One-Compartment Pharmacokinetic Model”, Talevi and Bellera [68].

$$\text{Definition total_conc } \{n\} \ (Ds : n.\text{-tuple } \mathbb{R}) : \mathbb{R} \rightarrow \mathbb{R} := \sum_{i < n} ((\text{cst } 0) \ \backslash \max (\text{Concentration } (\text{tnth } Ds \ i)) \ \backslash o (\text{shift } (-ttd * i \% R))) \quad \left| \quad C(\bar{D}, t) = \sum_{i=0}^n \max \left(0, \frac{\bar{D}_i \cdot k_a}{V_d \cdot (k_a - k_e)} \cdot (e^{-k_e \cdot (t-i \cdot ttd)} - e^{-k_a \cdot (t-i \cdot ttd)}) \right) \right.$$

Fig. 11. The function for the total concentration at time t , where Ds is an n tuple of all the doses given.

This is an over-approximation of the actual value of the concentration, because the exponential decay is only accounting for the concentration from each individual dose, and not the total concentration at any point in time. Because our safety property is expressed as *less than or equal to* a critical value, this approximation does not impede the safety of the system.

Finally, the desired safety-critical property can be expressed in Rocq, as shown in [Theorem 6.1](#):

THEOREM 6.1 (MAIN SAFETY THEOREM). *For any neural network $\mathcal{N}$, the concentration of the drug in a patient’s body never reaches a critical value.*

$$\text{Theorem doses_safe } (n : \text{nat}) \ (\text{initial } t : \mathbb{R}) \ (\text{initial_itv} : 0 \leq \text{initial} \leq C_safe) : \quad \left| \quad \forall \mathbf{x} \in \mathbb{R}^5, \forall t : \mathbb{R}, \right. \\ 0 \leq \sum_{i=0}^n \max \left(0, \frac{n_doses(\mathbf{x}_0, i) \cdot k_a}{V_d \cdot (k_a - k_e)} \cdot (e^{-k_e \cdot (t-i \cdot ttd)} - e^{-k_a \cdot (t-i \cdot ttd)}) \right) \leq C_safe$$

This theorem represents Φ in [Equation 1](#), and the remaining question is what should ψ be?

6.2 Neural Network Specification in VEHICLE

In order to derive the specification ψ that VEHICLE can verify, we restrict ttd to be greater than or equal to the time of the peak concentration. This restriction means that the concentration must be non-increasing at the point in time when a dose is given.⁵ Then, we find a specification of the network that is sufficient to prove [Theorem 6.1](#). Because, when the neural network is applied, the concentration is non-increasing, we can consider only the concentration from all previous doses at that point. Then, we consider the output of the network as a new dose being given and compute its peak. If the sum of the current concentration and calculated max concentration are less than or equal to C_safe , then it follows that at no point could the concentration ever be greater than C_safe . This constitutes our property ψ defined as

$$\forall \mathbf{x} \in \mathbb{R}^5, C(t) + C \left(\mathcal{N}(\mathbf{x}), \frac{\ln \left(\frac{k_a}{k_e} \right)}{k_a - k_e} \right) \leq C_safe, \quad (2)$$

⁵In real life application, it is plausible that the application of neural networks can be constrained in this way. This restriction allows the assumption that the concentration at a dosing point is higher than any future point.

Listing 3 The VEHICLE specification file, expressing the neural network safety property.

```

1  type Input = Tensor Real [5] 34  @property
2                                     35  C_safe_pos : Bool
3  conc = 0                        36  C_safe_pos = 0 < C_safe
4  temp = 1                       37
5  wbc = 2                        38  @property
6  age = 3                       39  ttd_pos : Bool
7  weight = 4                    40  ttd_pos = 0 < ttd
8                                     41
9  type Output = Tensor Real [1] 42  safeInput : Input -> Bool
10                                     43  safeInput x =
11  @network                       44  0 <= x ! conc <= C_safe and
12  pk : Input -> Output           45  36.5 <= x ! temp <= 40 and
13                                     46  7.5 <= x ! wbc <= 20 and
14  @parameter                     47  18 <= x ! age <= 89 and
15  Ka, Ke, Vd, C_safe, ttd,       48  50 <= x ! weight <= 100
16  Ka_over, Ka_under,            49
17  Ke_over, Ke_under : Real      50  safeOutput : Input -> Bool
18                                     51  safeOutput x =
19  @property                       52  let y = (((normpk x) ! 0)*Ka) / (Vd*(Ka - Ke)))
20  Ka_pos : Bool                  53  in if Ka < Ke
21  Ka_pos = 0 < Ka                54  then (x!conc) + y*(Ke_under - Ka_over) <= C_safe
22                                     55  else (x!conc) + y*(Ke_over - Ka_under) <= C_safe
23  @property                       56
24  Ke_pos : Bool                  57  @property
25  Ke_pos = 0 < Ke                58  safe : Bool
26                                     59  safe = forall x . safeInput x => safeOutput x
27  @property                       60
28  Ke_n_Ka : Bool                 61  nonNegOutput : Input -> Bool
29  Ke_n_Ka = Ka != Ke             62  nonNegOutput x = 0 <= (normpk x) ! 0
30                                     63
31  @property                       64  @property
32  Vd_pos : Bool                  65  nonNeg : Bool
33  Vd_pos = 0 < Vd                66  nonNeg = forall x . safeInput x => nonNegOutput x

```

where $C(t)$ is the initial total concentration when the dose is given. The VEHICLE spec for ψ is shown in Listing 3 where property safe encodes Equation 2. As peak time is a potentially-irrational number, K_{e_over} , K_{e_under} , K_{a_over} and K_{a_under} are approximations of the exponentials in Figure 10.

With these, VEHICLE can be used to verify ψ^6 by invoking the following command:

```
vehicle verify -v Marabou -c cache -s pk.vcl -n pk:pk.onnx -p Ka:3.5 -p Ke:4.5 ...,
```

where the `-p` arguments set the values for the parameters in the specification and the `-c` argument, which creates a cache for storing the parameter values and verification result for later retrieval.

Our specification is parametric in the patient-specific variables, such as k_a and k_e . In fact, the proof in Rocq only relies on the parameters being positive and $k_a \neq k_e$. These restrictions are expressed as properties in VEHICLE.⁷ This is a generalisation of the approach taken in the Wind-Controller example presented earlier, where we assumed that system parameters such as the sensor

⁶Due to limitations in Marabou wrt. strict inequalities, adaptations to the specification have had to be made. The properties shown in the specification file are sufficient to prove the main theorem.

⁷Note that VEHICLE cannot verify that ttd is greater than or equal to the time of peak dose, because it is potentially irrational. Instead, the job of dealing with the irrational parameters is delegated to Rocq.

error would be the same for any car. Now our proofs are parametric on the patient's data. Crucially, the execution of the above command gives us a "Yes" answer: the neural network is proven safe by the neural solver Marabou. The specification then is extracted to Rocq using `vehicle export -t Rocq -c cache -o Spec.v -r`. This sets the target language to Rocq, the cache path, the output file and a Rocq specific flag `-r`, that causes VEHICLE to extract to the constructive reals as defined in the Rocq standard library, as opposed to the default MathComp `realType` interface. We obtain a file translating the neural network safety property of Listing 3 in Rocq, see Appendix B.

6.3 System Correctness

Next, we prove in Rocq that the properties verified by VEHICLE are sufficient to prove the main safety [Theorem 6.1](#):

THEOREM 6.2. *For any neural network $\mathcal{N}$, for any valid initial state $\mathbf{x}$ (omitted from the equation for brevity), ψ ([Equation 2](#)) implies an over-dose could never occur.*

Theorem doses_safe

```
(n : nat) (initial t : R)
(initial_itv : 0 <= initial <= C_safe) :
(forall C : R, 0 <= C <= C_safe ->
  C + (Concentration (network C) dCdt_root)
    <= C_safe)
(0 <= total_conc (n_doses initial n) t
  <= C_safe).
```

$\forall \mathbf{x} \in \mathbb{R}^5, \forall t : \mathbb{R},$

$$C(t) + C\left(\mathcal{N}(\mathbf{x}), \frac{\ln\left(\frac{k_a}{k_e}\right)}{k_a - k_e}\right) \leq C_{safe} \implies$$

$$0 \leq \sum_{i=0}^n \max\left(0, \frac{n_doses(\mathbf{x}_0, i) \cdot k_a}{V_d \cdot (k_a - k_e)} \cdot \left(e^{-k_e \cdot (t - i \cdot ttd)} - e^{-k_a \cdot (t - i \cdot ttd)}\right)\right) \leq C_{safe}$$

PROOF. We perform induction on n , the size of the tuple Ds from [Figure 11](#). For $n = 0$, it is trivial, as concentration is always 0. For the inductive case, we proceed by cases on t and $S(n) \cdot ttd$.

Case 1: $t \leq S(n) \cdot ttd$. The $S(n)^{\text{th}}$ dose has not been given or has been given at this instant, so it follows from the inductive hypothesis. This is shown by the blue line from 0 to 1 in [Figure 8](#).

Case 2: $S(n) \cdot ttd < t$ implies that the $S(n)^{\text{th}}$ dose has been given and had some time to be absorbed. We first note that the equation in [Figure 11](#) is differentiable for all t s.t. $S(n) \cdot ttd < t$, because $\max(x, y)$ is differentiable everywhere except $x = y$, and the n^{th} concentration function is strictly positive for all $t > S(n) \cdot ttd$. The sum can be split into two components, the concentration from all previous doses (the red line in [Figure 8](#)) and the newly added dose:

$$\sum_{i=0}^n C_i(t) + C_{n+1}(t) \leq C_{safe} \quad (3)$$

Note that $C_i(t)$ is equivalent to $C(D, t)$ where D is the i^{th} dose given to the patient.

Because $C_i(t)$ is decreasing for all $S(n) \cdot ttd < t$, $C_i(t) < C_i(S(n) \cdot ttd)$, we can weaken the statement. This proves that the blue dashed line is always greater than or equal to the red line in [Figure 8](#).

$$\sum_{i=0}^n C_i(S(n) \cdot ttd) + C_{n+1}(t) \leq C_{safe} \quad (4)$$

At $S(n) \cdot ttd$, the new concentration has no effect, so the summation is simply the total concentration, $C(S(n) \cdot ttd)$:

$$C(S(n) \cdot ttd) + C_{n+1}(t) \leq C_{safe} \quad (5)$$

As $\frac{\ln(\frac{k_a}{k_e})}{k_a - k_e}$, the root of the derivative, is the maximum of the function, we can similarly weaken this:

$$C(S(n) \cdot ttd) + C_{n+1} \left(\frac{\ln(\frac{k_a}{k_e})}{k_a - k_e} \right) \leq C_{safe} \quad (6)$$

Notice how the left side of the + is exactly the concentration at the point the dose is given, and the right side is exactly the peak concentration of our new dose. This is exactly [Equation 2](#)! We have successfully both derived a property ψ that `VEHICLE` can verify, and shown that $\psi(\mathcal{N}) \implies \Phi(S(\mathcal{N}))$.⁸ $\square$

The resulting Rocq proof (submitted in supplementary materials [22]) is ~1200 LOC, and many parts of both MathComp and Analysis are leveraged to complete it. Derivatives are used to reason about the peak concentration, continuity is used to reason about the main functions. As part of this proof, we contributed an original addition to MathComp Analysis, the proof that $f \setminus \max g$ is Gateaux derivable at x if f and g are Gateaux derivable and continuous at x , and $f(x) \neq g(x)$. The following fact implies the derivability of $\max$ and $\min$, as well as lemmas like `derive_max1`.⁹

```
Fact der_max f g x v :
  f x <> g x -> derivable f x v -> derivable g x v ->
  {for x, continuous f} -> {for x, continuous g} ->
  (fun h => h^-1 *: (((f \max g) \o shift x) (h *: v) - (f \max g) x)) @
  0^1 --> if f x < g x then 'D_v g x else 'D_v f x.
```

```
Lemma derive_max1 f g x v : f x > g x ->
  {for x, continuous f} -> {for x, continuous g} -> 'D_v (f \max g) x = 'D_v f x.
```

In addition, `interval` was used to reason about the parameters in the extract Rocq file, to prove:

$$K_{e_under} \leq e^{-k_e \frac{\ln(\frac{k_a}{k_e})}{k_a - k_e}} \leq K_{e_under}, \quad K_{a_under} \leq e^{-k_a \frac{\ln(\frac{k_a}{k_e})}{k_a - k_e}} \leq K_{a_under}$$

To show that the specification is satisfiable, we constructed a simple ReLU network with 5 inputs, 2 hidden layers of 128 and 64 nodes respectively, and 1 output. We use the `VEHICLE` Machine Learning backend to train the network using the specification. The resulting network is verified to satisfy the safety properties laid out in [Listing 3](#) using the `VEHICLE` solver backend and is available alongside the specification in the supplementary material. Accordingly, all backends of `VEHICLE` were used in this case study.

In summary, this case study serves as an example of how the proposed compositional approach enables infinite-horizon safety verification in domains where the system dynamics is continuous. [Table 3](#) gives a comparison between this case study and the *Wind Controller* example we considered in the previous sections. Both case studies showed the power of compositional approach to NCPS verification and inductive proofs of infinite-horizon safety properties. However, the *Medical Controller* example addresses a continuous system by leveraging existing mathematical libraries inside the ITPs. We believe that the presented method is applicable to exponential decay problems more generally and showcases how a neural network acting in a discrete capacity can be verified in a continuous system.

A natural question is whether this case study could be replicated in other ITPs. While we have implemented Vehicle integration for Agda, Isabelle/HOL and Imandra, the existence of the lemmas from `CoqInterval`, `MathComp`, and `MathComp analysis` used in this proof within each ITP is an

⁸Because the time of peak concentration is potentially irrational, an over-approximation is taken by `VEHICLE`.

⁹Our proof of `der_max` and the respective lemmas have been merged into MathComp Analysis.

Table 3. Comparison of the *Wind Controller* case study and the *Medical Controller* case study.

NCPS Model \ Safety Proofs	Wind Controller	Medical Controller
NCPS composes system and neural levels	✓	✓
Controller trained via <code>VEHICLE</code> ML backend	✗	✓
Entire system description given in ITP	✓	✓
Infinite horizon modelling via inductive proofs	✓	✓
Proofs concern complex continuous system dynamics	✗	✓
Proofs rely on larger mathematical library infrastructure	✗	✓

orthogonal problem. Not all ITPs are equal in this regard: they do not exist within Agda, but are already largely present in Isabelle/HOL. Section 5.4 discusses necessary steps to formalize such libraries in an ITP from scratch using Imandra as an example.

`VEHICLE`'s design provides a uniform interface for neural network verification that is agnostic to the chosen ITP and symbolic CPS formalisation. Consequently, `VEHICLE` can integrate with all existing work on formalising CPS systems in ITPs, regardless of whether the symbolic system is formalized via (temporal) differential dynamic logic in Isabelle/HOL and Rocq [14] or PVS [77], via hybrid predicate transformers in Isabelle/HOL [33, 80, 81], via predicate abstraction in Rocq [36], or in another framework [15, 55, 59]. We envisage users building on this substantial body of research in a manner that is best suited to their case study.

7 Conclusions

Verification of neural-cyber-physical systems is a promising application area for ITPs, but requires integrated reasoning about the neural component and the symbolic cyber-physical components. Our approach addresses the current limitations of the state-of-the-art tools [5, 13, 37, 39, 76] participating in ARCH-COMP [34]. Since these tools are based on reachability methods, they can typically only handle safety proofs for finite time horizons, rarely exceeding a few hundred time steps. In contrast, in this paper we have made a fundamental shift from finite to infinite-time horizon guarantees, (i.e. guarantees that keep the system safe at *any* future point in time). Concretely we have shown how our Haskell DSL `VEHICLE` provides the first ever practical, fully compositional approach to integrating proofs about the neural components from neural solvers into symbolic proofs about the cyber-physical components in ITPs. This paper complements the existing solver [24, 25] and machine-learning backends of `VEHICLE` with an original technical description of the core type-checker and its ITP backend.

We have proven that our approach allows lightweight integration with a diverse range of ITPs with and without dependent types, by integrating `VEHICLE` with Agda, Rocq, Isabelle/HOL and Imandra. On average this integration only required 775 lines of ITP code per ITP, although in Isabelle/HOL where formalisations of real numbers and tensors already exist, integration was achieved in as low as 178 lines. The biggest per-ITP effort was the development of a new tensor library for MathComp, – also an original contribution of this paper. To show the benefits of `VEHICLE`'s compositional approach, we then used this library in the verification of the infinite time-horizon safety of a continuous NCPS in Rocq's MathComp, – to our knowledge this is the first result of this kind in any ITP, and a testament to the power of compositionality in the new emerging application domain of neural-cyber-physical system verification.

8 Other Related and Future Work

Other functional DSLs for neural network verification. The tool CAISAR [4], written in Why3 and OCAML, offers a DSL for high-level neural network specifications; and compiles those specifications to a range of neural solvers. However, it has no capacity for building ITP backends, and thus does not facilitate NCPS verification tasks.

Certificates for Neural Networks. Currently the validity of the neural specifications generated by Vehicle are not checked by the ITP core, and therefore the ITP must trust Vehicle and the neural solvers. We believe that recent efforts to extend neural solvers with proof production [38] and to check such proofs in theorem provers [29] can be seamlessly integrated with VEHICLE in future. For example, in the Isabelle/HOL backend described in Section 5.3, the locales generated by Vehicle can be instantiated by a function and a proof that this concrete function satisfies the locale's assumptions. Given an imported proof from a neural solver, it would be possible to obtain all results proven within the locale, now *instantiated* for the concrete definition and checkable by Isabelle itself.

Neural Networks as Certificates. An alternative to verifying reachability properties is to learn neural network-based certificates [1, 7, 18, 27], such as barrier functions or Lyapunov functions, satisfaction of which provides formal guarantees about the NCPS. While these approaches provide infinite-time horizon guarantees, the central component of their safety argument, e.g. encoded in the neural network-based barrier functions, is *entirely sub-symbolic*, and itself unverified. We believe that VEHICLE and ITPs may be of use here.

Neural networks and ODEs. Many works have looked at applying neural networks to cyber-physical system verification. Theoretically, reachability analysis can be combined with ODE solvers by expressing ODEs as neural networks [82] or neural networks as ODEs [39]. In practice, the resulting approaches suffer from the limitations of automated reachability-based techniques discussed in Section 1 (in particular state space complexity and time horizons).

New Applications. Finally, we plan to apply VEHICLE to further real-life application in the NCPS domain. For example, this paper does not re-implement prior ARCH-COMP benchmark translations as these require non-linear neural network specifications. While such specifications are already supported by VEHICLE, existing non-linear solvers [71] are not yet compatible with the official VNN-LIB 2.0 standard [61] which is a prerequisite for integration with VEHICLE.

Contribution Statement

Daggitt: Conceptualisation, Type system, Agda backend. **Komendantskaya:** Conceptualisation, Medical case study. **Sirman:** Medical case study, Rocq backend. **Teuber:** Isabelle backend, CPS discussion. **Bruni:** Rocq backend, Rocq tensor library, Medical case study. **Smart:** Rocq backend, Rocq tensor library. **Passmore:** Imandra backend.

Data-Availability Statement

The VEHICLE source code, the medical case study source code, and a virtual machine containing Vehicle and the case study, along side all necessary packages, can be found in Daggitt et al. [22].

Acknowledgments

E.Komendantskaya acknowledges the support of the grant UKRI AI Centre for Doctoral Training in Dependable and Deployable Artificial Intelligence for Robotics (CDT-D2AIR). The PhD scholarship of A.Sirman was funded by EPSRC DLA Collaborative Studentship.

References

- [1] Alessandro Abate, Daniele Ahmed, Mirco Giacobbe, and Andrea Peruffo. 2021. Formal Synthesis of Lyapunov Neural Networks. *IEEE Control. Syst. Lett.* 5, 3 (2021), 773–778. doi:10.1109/LCSYS.2020.3005328
- [2] Reynald Affeldt, Yves Bertot, Alessandro Bruni, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, Pierre Roux, Kazuhiko Sakaguchi, Zachary Stone, Pierre-Yves Strub, and Laurent Théry. 2026. *MathComp-Analysis: Mathematical Components compliant analysis library*. <https://github.com/math-comp/analysis> original-date: 2017-11-29T12:17:44Z.
- [3] Aws Albarghouthi. 2021. Introduction to Neural Network Verification. *Found. Trends Program. Lang.* 7, 1-2 (2021), 1–157. doi:10.1561/25000000051
- [4] Michele Alberti, François Bobot, Julien Girard-Satabin, Alban Grastien, Aymeric Varasse, and Zakaria Chihani. 2025. The CAISAR Platform: Extending the Reach of Machine Learning Specification and Verification. In *Integrated Formal Methods - 20th International Conference, iFM 2025, Paris, France, November 19-21, 2025, Proceedings (LNCS, Vol. 16194)*, Ferruccio Damiani and Marie Farrell (Eds.). Springer, Cham, Switzerland, 290–309. doi:10.1007/978-3-032-10794-7_15
- [5] Matthias Althoff. 2015. An Introduction to CORA 2015. In *Proc. of the 1st and 2nd Workshop on Applied Verification for Continuous and Hybrid Systems*. EasyChair, 120–151. doi:10.29007/zbkv
- [6] Jason Ansel, Edward Z. Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, et al. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafir (Eds.). ACM, New York, NY, USA, 929–947. doi:10.1145/3620665.3640366
- [7] Edoardo Bacci, Mirco Giacobbe, and David Parker. 2021. Verifying Reinforcement Learning up to Infinity. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, Zhi-Hua Zhou (Ed.). ijcai.org, 2154–2160. doi:10.24963/IJCAI.2021/297
- [8] Alexander Bagnall and Gordon Stewart. 2019. Certifying the True Error: Machine Learning in Coq with Verified Generalization Guarantees. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, 2662–2669. doi:10.1609/AAAI.V33I01.33012662
- [9] Stanley Bak, Hoang-Dung Tran, Kerianne Hobbs, and Taylor T. Johnson. 2020. Improved Geometric Path Enumeration for Verifying ReLU Neural Networks. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I (LNCS, Vol. 12224)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, Cham, Switzerland, 66–96. doi:10.1007/978-3-030-53288-8_4
- [10] Haniel Barbosa, Chantal Keller, Andrew Reynolds, Arjun Viswanathan, Cesare Tinelli, and Clark W. Barrett. 2023. An Interactive SMT Tactic in Coq using Abductive Reasoning. In *LPAR 2023: Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Manizales, Colombia, 4-9th June 2023 (EPiC Series in Computing, Vol. 94)*, Ruzica Piskac and Andrei Voronkov (Eds.). EasyChair, 11–22. doi:10.29007/432M
- [11] Alexander Bentkamp. 2016. Expressiveness of Deep Learning. *Archive of Formal Proofs* (November 2016). https://isa-afp.org/entries/Deep_Learning.html, Formal proof development.
- [12] Errett Bishop, D. S. Bridges, and Errett Bishop. 1985 - 1985. *Constructive analysis / Errett Bishop, Douglas Bridges*. Springer-Verlag, Berlin .
- [13] Sergiy Bogomolov, Marcelo Forets, Goran Frehse, Kostiantyn Potomkin, and Christian Schilling. 2019. JuliaReach: a toolbox for set-based reachability. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, Montreal, QC, Canada, April 16-18, 2019*, Necmiye Ozay and Pavithra Prabhakar (Eds.). ACM, New York, NY, USA, 39–44. doi:10.1145/3302504.3311804
- [14] Rose Bohrer, Vincent Rahli, Ivana Vukotic, Marcus Völz, and André Platzer. 2017. Formally verified differential dynamic logic. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, Yves Bertot and Viktor Vafeiadis (Eds.). ACM, New York, NY, USA, 208–221. doi:10.1145/3018610.3018616
- [15] Robert S Boyer, Milton W Green, and J Strother Moore. 1990. The use of a formal simulator to verify a simple real time control program. In *Beauty is Our Business: A Birthday Salute to Edsger W. Dijkstra*. Springer, Berlin, Heidelberg, 54–66.
- [16] Achim D. Brucker and Amy Stell. 2023. Verifying Feedforward Neural Networks for Classification in Isabelle/HOL. In *Formal Methods - 25th International Symposium, FM 2023, Lübeck, Germany, March 6-10, 2023, Proceedings (LNCS, Vol. 14000)*, Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker (Eds.). Springer, Cham, Switzerland, 427–444. doi:10.1007/978-3-031-27481-7_24
- [17] Alessandro Bruni. 2026. Tensor formalization #1535. GitHub Pull Request. <https://github.com/math-comp/math-comp/pull/1535> Mathematical Components library.
- [18] Ya-Chien Chang, Nima Roohi, and Sicun Gao. 2019. Neural Lyapunov Control. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B.

- Fox, and Roman Garnett (Eds.). Curran Associates, Inc., Red Hook, NY, USA, 3240–3249. <https://proceedings.neurips.cc/paper/2019/hash/2647c1dba23bc0e0f9cdf75339e120d2-Abstract.html>
- [19] Cyril Cohen and Damien Rouhling. 2017. A Formal Proof in Coq of LaSalle’s Invariance Principle. In *Interactive Theorem Proving (ITP) (2017) (LNCS, Vol. 10499)*. Springer, 148–163. doi:10.1007/978-3-319-66107-0_10
 - [20] Cyril Cohen, Pierre Roux, Enrico Tassi, Kazuhiko Sakaguchi, Reynald Affeldt, Laurent Théry, Erik Martin-Dorel, Georges Gonthier, et al. 2024. *math-comp/math-comp: The Mathematical Components Library 2.3.0*. doi:10.5281/zenodo.14237175
 - [21] Lucas C. Cordeiro, Matthew L. Daggitt, Julien Girard-Satabin, Omri Isac, Taylor T. Johnson, Guy Katz, Ekaterina Komendantskaya, Augustin Lemesle, Edoardo Manino, Artjoms Sinkarovs, and Haoze Wu. 2025. Neural Network Verification is a Programming Language Challenge. In *Programming Languages and Systems - 34th European Symposium on Programming, ESOP 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part I (LNCS, Vol. 15694)*, Viktor Vafeiadis (Ed.). Springer, Cham, Switzerland, 206–235. doi:10.1007/978-3-031-91118-7_9
 - [22] Matthew Daggitt, Ekaterina Komendantskaya, Alistair Sirman, Alessandro Bruni, Samuel Teuber, Josh Smart, and Grant Passmore. 2026. *Compositional Neural-Cyber-Physical System Verification in the Interactive Theorem Prover of Your Choice Artefact*. doi:10.5281/zenodo.20529848
 - [23] Matthew L. Daggitt, Guillaume Allais, James McKinna, Andreas Abel, Nathan Van Doorn, James Wood, Ulf Norell, Donnacha Oisín Kidney, et al. 2025. The Agda Standard Library: Version 2.0. *Journal of Open Source Software* 10, 116 (Dec. 2025), 9241. doi:10.21105/joss.09241
 - [24] Matthew L. Daggitt, Robert Atkey, Wen Kokke, Ekaterina Komendantskaya, and Luca Arnaboldi. 2023. Compiling Higher-Order Specifications to SMT Solvers: How to Deal with Rejection Constructively. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*, Robbert Krebbers, Dmitriy Traytel, Brigitte Pientka, and Steve Zdancewic (Eds.). ACM, New York, NY, USA, 102–120. doi:10.1145/3573105.3575674
 - [25] Matthew L. Daggitt, Wen Kokke, and Robert Atkey. 2024. Efficient compilation of expressive problem space specifications to neural network solvers. *CoRR abs/2402.01353* (2024). arXiv:2402.01353 doi:10.48550/ARXIV.2402.01353
 - [26] Matthew L. Daggitt, Wen Kokke, Robert Atkey, Ekaterina Komendantskaya, Natalia Slusarz, and Luca Arnaboldi. 2025. Vehicle: Bridging the Embedding Gap in the Verification of Neuro-Symbolic Programs (Invited Talk). In *10th International Conference on Formal Structures for Computation and Deduction, FSCD 2025, Birmingham, UK, July 14-20, 2025 (LIPIcs, Vol. 337)*, Maribel Fernández (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:20. doi:10.4230/LIPICS.FSCD.2025.2
 - [27] Charles Dawson, Zengyi Qin, Sicun Gao, and Chuchu Fan. 2021. Safe Nonlinear Control Using Robust Neural Lyapunov-Barrier Functions. In *Conference on Robot Learning, 8-11 November 2021, London, UK (Proceedings of Machine Learning Research, Vol. 164)*, Aleksandra Faust, David Hsu, and Gerhard Neumann (Eds.). PMLR, 1724–1735. <https://proceedings.mlr.press/v164/dawson22a.html>
 - [28] Leonardo Mendonça de Moura and Grant Olney Passmore. 2013. Computation in Real Closed Infinitesimal and Transcendental Extensions of the Rationals. In *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings (LNCS, Vol. 7898)*, Maria Paola Bonacina (Ed.). Springer, Berlin, Heidelberg, 178–192. doi:10.1007/978-3-642-38574-2_12
 - [29] Remi Desmartin, Omri Isac, Grant O. Passmore, Ekaterina Komendantskaya, Kathrin Stark, and Guy Katz. 2025. A Certified Proof Checker for Deep Neural Network Verification in Imandra. In *16th International Conference on Interactive Theorem Proving, ITP 2025, Reykjavik, Iceland, September 28 - October 1, 2025 (LIPIcs, Vol. 352)*, Yannick Forster and Chantal Keller (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1:1–1:21. doi:10.4230/LIPICS.ITP.2025.1
 - [30] Remi Desmartin, Grant O. Passmore, Ekaterina Komendantskaya, and Matthew L. Daggitt. 2022. 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*. ACM, New York, NY, USA, 3:1–3:14. doi:10.1145/3551357.3551372
 - [31] Kees Dullemond and Kasper Peeters. 1991. Introduction to tensor calculus. *Kees Dullemond and Kasper Peeters* (1991), 42–44.
 - [32] Marc Fischer, Mislav Balunovic, Dana Drachler-Cohen, Timon Gehr, Ce Zhang, and Martin T. Vechev. 2019. DL2: Training and Querying Neural Networks with Logic. In *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)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 1931–1941.
 - [33] Simon Foster, Jonathan Julián Huerta y Munive, Mario Gleirscher, and Georg Struth. 2021. Hybrid Systems Verification with Isabelle/HOL: Simpler Syntax, Better Models, Faster Proofs. In *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings (LNCS, Vol. 13047)*, Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan (Eds.). Springer, Cham, Switzerland, 367–386. doi:10.1007/978-3-030-90870-6_20

- [34] Goran Frehse and Matthias Althoff (Eds.). 2025. *Proceedings of 12th Int. Workshop on Applied Verification for Continuous and Hybrid Systems*. EPIC Series in Computing, Vol. 108. EasyChair.
- [35] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völz, and André Platzer. 2015. KeYmaera X: An Axiomatic Tactical Theorem Prover for Hybrid Systems. In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings (LNCS, Vol. 9195)*, Amy P. Felty and Aart Middeldorp (Eds.). Springer, Cham, Switzerland, 527–538. doi:10.1007/978-3-319-21401-6_36
- [36] Herman Geuvers, Adam Koprowski, Dan Synek, and Eelis van der Weegen. 2010. Automated Machine-Checked Hybrid System Safety Proofs. In *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings (LNCS, Vol. 6172)*, Matt Kaufmann and Lawrence C. Paulson (Eds.). Springer, Berlin, Heidelberg, 259–274. doi:10.1007/978-3-642-14052-5_19
- [37] Chao Huang, Jiameng Fan, Xin Chen, Wenchao Li, and Qi Zhu. 2022. POLAR: A Polynomial Arithmetic Framework for Verifying Neural-Network Controlled Systems. In *Automated Technology for Verification and Analysis - 20th International Symposium, ATVA 2022, Virtual Event, October 25-28, 2022, Proceedings (LNCS, Vol. 13505)*, Ahmed Bouajjani, Lukás Holík, and Zhilin Wu (Eds.). Springer, Cham, Switzerland, 414–430. doi:10.1007/978-3-031-19992-9_27
- [38] Omri Isac, Clark W. Barrett, Min Zhang, and Guy Katz. 2022. Neural Network Verification with Proof Production. In *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, Alberto Griggio and Neha Rungta (Eds.). IEEE, Piscataway, NJ, USA, 38–48. doi:10.34727/2022/ISBN.978-3-85448-053-2_9
- [39] Radoslav Ivanov, Taylor J. Carpenter, James Weimer, Rajeev Alur, George J. Pappas, and Insup Lee. 2021. Verisig 2.0: Verification of Neural Network Controllers Using Taylor Model Preconditioning. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I (LNCS, Vol. 12759)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, Cham, Switzerland, 249–262. doi:10.1007/978-3-030-81685-8_11
- [40] Mark P Jones. 2003. *Qualified types: theory and practice*. Number 9. Cambridge University Press, Cambridge, UK.
- [41] Kyle D. Julian, Mykel J. Kochenderfer, and Michael P. Owen. 2019. Deep Neural Network Compression for Aircraft Collision Avoidance Systems. *Journal of Guidance, Control, and Dynamics* 42, 3 (2019), 598–608. doi:10.2514/1.G003724
- [42] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I (LNCS, Vol. 10426)*, Rupak Majumdar and Viktor Kuncak (Eds.). Springer, Cham, Switzerland, 97–117. doi:10.1007/978-3-319-63387-9_5
- [43] Colin Kessler, Ekaterina Komendantskaya, Marco Casadio, Ignazio Maria Viola, Thomas Flinkow, Albaraa Ammar Othman, Alistair Malhotra, and Robbie McPherson. 2025. Neural Network Verification for Gliding Drone Control: A Case Study. In *AI Verification - Second International Symposium, SAIV 2025, Zagreb, Croatia, July 21-22, 2025, Proceedings (LNCS, Vol. 15947)*, Mirco Giacobbe and Anna Lukina (Eds.). Springer, Cham, Switzerland, 180–199. doi:10.1007/978-3-031-99991-8_9
- [44] Hanna Lachnitt, Mathias Fleury, Haniel Barbosa, Jibiana Jakpor, Bruno Andreotti, Andrew Reynolds, Hans-Jörg Schurr, Clark W. Barrett, and Cesare Tinelli. 2025. Improving the SMT Proof Reconstruction Pipeline in Isabelle/HOL. In *16th International Conference on Interactive Theorem Proving, ITP 2025, Reykjavik, Iceland, September 28 - October 1, 2025 (LIPIcs, Vol. 352)*, Yannick Forster and Chantal Keller (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 26:1–26:22. doi:10.4230/LIPICS.ITP.2025.26
- [45] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers (LNCS, Vol. 6355)*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer, Berlin, Heidelberg, 348–370. doi:10.1007/978-3-642-17511-4_20
- [46] Andres Löb, Conor McBride, and Wouter Swierstra. 2010. A Tutorial Implementation of a Dependently Typed Lambda Calculus. *Fundam. Informaticae* 102, 2 (2010), 177–207. doi:10.3233/FI-2010-304
- [47] Diego Manzananas Lopez, Sung Woo Choi, Hoang-Dung Tran, and Taylor T. Johnson. 2023. NNV 2.0: The Neural Network Verification Tool. In *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II (LNCS, Vol. 13965)*, Constantin Enea and Akash Lal (Eds.). Springer, Cham, Switzerland, 397–412. doi:10.1007/978-3-031-37703-7_19
- [48] Diego Manzananas Lopez, Patrick Musau, Hoang-Dung Tran, Souradeep Dutta, Taylor J. Carpenter, Radoslav Ivanov, and Taylor T. Johnson. 2019. ARCH-COMP19 Category Report: Artificial Intelligence and Neural Network Control Systems (AINNCS) for Continuous and Hybrid Systems Plants. In *ARCH19. 6th International Workshop on Applied Verification of Continuous and Hybrid Systems (EPIC Series in Computing, Vol. 61)*, Goran Frehse and Matthias Althoff (Eds.). EasyChair, 103–119. doi:10.29007/rgv8
- [49] Ignacio D. Lopez-Miguel, Borja Fernández Adiego, Faiq Ghawash, and Enrique Blanco Viñuela. 2023. Verification of Neural Networks Meets PLC Code: An LHC Cooling Tower Control System at CERN. In *Engineering Applications of Neural Networks - 24th International Conference, EAAAI/EANN 2023, León, Spain, June 14-17, 2023, Proceedings (Communications in Computer and Information Science, Vol. 1826)*, Lazaros Iliadis, Ilias Maglogiannis, Serafin Alonso,

- Chrisina Jayne, and Elias Pimenidis (Eds.). Springer, Cham, Switzerland, 420–432. doi:10.1007/978-3-031-34204-2_35
- [50] Érik Martin-Dorel and Guillaume Melquiond. 2016. Proving Tight Bounds on Univariate Expressions with Elementary Functions in Coq. *J. Autom. Reason.* 57, 3 (2016), 187–217. doi:10.1007/S10817-015-9350-4
- [51] J. Garrett Morris and Mark P. Jones. 2010. Instance chains: type class programming without overlapping instances. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, New York, NY, USA, 375–386. doi:10.1145/1863543.1863596
- [52] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. LNCS, Vol. 2283. Springer, Berlin, Heidelberg. doi:10.1007/3-540-45949-9
- [53] Ulf Norell. 2009. Dependently typed programming in Agda. In *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, Andrew Kennedy and Amal Ahmed (Eds.). ACM, New York, NY, USA, 1–2. doi:10.1145/1481861.1481862
- [54] Russell O'Connor. 2009. *Incompleteness and Completeness: Formalizing Logic and Analysis in Type Theory*. Ph.D. Dissertation. Radboud University Nijmegen.
- [55] Samir Ouchani, Khaled Khebbab, and Meriem Hafsi. 2020. Towards Enhancing Security and Resilience in CPS: A Coq-Maude based Approach. In *17th IEEE/ACS International Conference on Computer Systems and Applications, AICCSA 2020, Antalya, Turkey, November 2-5, 2020*. IEEE, Piscataway, NJ, USA, 1–6. doi:10.1109/AICCSA50499.2020.9316535
- [56] Grant O. Passmore, Simon Cruanes, Denis Ignatovich, Dave Aitken, Matt Bray, Elijah Kagan, Kostya Kanishev, Ewen Maclean, and Nicola Mometto. 2020. The Imandra Automated Reasoning System (System Description). In *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II (LNCS, Vol. 12167)*, Nicolas Peltier and Viorica Sofronie-Stokkermans (Eds.). Springer, Cham, Switzerland, 464–471. doi:10.1007/978-3-030-51054-1_30
- [57] Ethan A Poweleit, Alexander A Vinks, and Tomoyuki Mizuno. 2023. Artificial intelligence and machine learning approaches to facilitate therapeutic drug management and model-informed precision dosing. *Therapeutic drug monitoring* 45, 2 (2023), 143–150.
- [58] Yicheng Qian, Joshua Clune, Clark W. Barrett, and Jeremy Avigad. 2025. Lean-Auto: An Interface Between Lean 4 and Automated Theorem Provers. In *Computer Aided Verification - 37th International Conference, CAV 2025, Zagreb, Croatia, July 23-25, 2025, Proceedings, Part III (LNCS, Vol. 15933)*, Ruzica Piskac and Zvonimir Rakamaric (Eds.). Springer, Cham, Switzerland, 175–196. doi:10.1007/978-3-031-98682-6_10
- [59] Daniel Ricketts. 2017. *Verification of Sampled-Data Systems using Coq*. Ph.D. Dissertation. University of California, San Diego, USA. <http://www.escholarship.org/uc/item/5n1899s2>
- [60] Damien Rouhling. 2018. A Formal Proof in Coq of a Control Function for the Inverted Pendulum. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP) (2018)*. 28–41. doi:10.1145/3167101
- [61] A. Roy, A. Antony, A. Gimelli, and M.L. Daggitt. 2026. VNN-LIB 2.0: Rigorous Foundations for Neural Network Verification. In *Computer Aided Verification, CAV 2026*.
- [62] Zhouxing Shi, Qirui Jin, Zico Kolter, Suman Jana, Cho-Jui Hsieh, and Huan Zhang. 2025. Neural Network Verification with Branch-and-Bound for General Nonlinearities. In *Tools and Algorithms for the Construction and Analysis of Systems - 31st International Conference, TACAS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part I (LNCS, Vol. 15696)*, Arie Gurfinkel and Marijn Heule (Eds.). Springer, Cham, Switzerland, 315–335. doi:10.1007/978-3-031-90643-5_17
- [63] George F. Simmons. 1972. *Differential Equations with Applications and Historical Notes* (3 ed.). CRC Press.
- [64] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin T. Vechev. 2019. An abstract domain for certifying neural networks. *Proc. ACM Program. Lang.* 3, POPL (2019), 41:1–41:30. doi:10.1145/3290354
- [65] Natalia Slusarz, Ekaterina Komendantskaya, Matthew L. Daggitt, and Robert J. Stewart. 2022. Differentiable Logics for Neural Network Training and Verification. In *15th International Workshop, Numerical Software Verification 2022, Haifa, Israel, July 31 - August 1, and August 11, 2022, Proceedings (LNCS, Vol. 13466)*, Omri Isac, Radoslav Ivanov, Guy Katz, Nina Narodytska, and Laura Nenzi (Eds.). Springer, Cham, Switzerland, 67–77. doi:10.1007/978-3-031-21222-2_5
- [66] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent types and multi-monadic effects in F. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodik and Rupak Majumdar (Eds.). ACM, New York, NY, USA, 256–270. doi:10.1145/2837614.2837655
- [67] Wouter Swierstra. 2008. Data types à la carte. *J. Funct. Program.* 18, 4 (2008), 423–436. doi:10.1017/S0956796808006758
- [68] Alan Talevi and Carolina L. Bellera. 2021. One-Compartment Pharmacokinetic Model. In *The ADME Encyclopedia: A Comprehensive Guide on Biopharmacy and Pharmacokinetics*. Springer International Publishing, Cham, Switzerland, 1–8. doi:10.1007/978-3-030-51519-5_58-1
- [69] Alan Talevi and Carolina L. Bellera. 2021. Two-Compartment Pharmacokinetic Model. In *The ADME Encyclopedia*. Springer, Cham, Switzerland, 1–8. doi:10.1007/978-3-030-84860-6_59

- [70] Samuel Teuber, Debasmita Lohar, and Bernhard Beckert. 2025. Of Good Demons and Bad Angels: Guaranteeing Safe Control under Finite Precision. In *Proceedings of the 25th Conference on Formal Methods in Computer-Aided Design, FMCAD 2025, Menlo Park, CA, USA, October 6-10, 2025*, Ahmed Irfan and Daniela Kaufmann (Eds.). TU Wien Academic Press, Vienna, Austria. doi:10.34727/2025/ISBN.978-3-85448-084-6_12
- [71] Samuel Teuber, Stefan Mitsch, and André Platzer. 2024. Provably Safe Neural Network Controllers via Differential Dynamic Logic. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.). Curran Associates, Inc., Red Hook, NY, USA, 1586–1624. http://papers.neurips.cc/paper_files/paper/2024/hash/031b5fd7d847f69ed33378a9a1117b4b-Abstract-Conference.html
- [72] The Rocq Development Team. 2025. *The Rocq Reference Manual, version 9.1.0*.
- [73] Holger Thies. 2026. Computing Solutions for Systems of Multivariate Ordinary Differential Equations in Rocq. In *Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP) (2026)*. 29–44. doi:10.1145/3779031.3779097
- [74] Niki Vazou. 2016. *Liquid Haskell: Haskell as a Theorem Prover*. Ph. D. Dissertation. University of California, San Diego, USA.
- [75] Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and J. Zico Kolter. 2021. Beta-CROWN: Efficient Bound Propagation with Per-neuron Split Constraints for Neural Network Robustness Verification. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, Marc’Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan (Eds.). Curran Associates, Inc., Red Hook, NY, USA, 29909–29921. <https://proceedings.neurips.cc/paper/2021/hash/fac7fead96dafceaf80c1daffae82a4-Abstract.html>
- [76] Yixuan Wang, Weichao Zhou, Jiameng Fan, Zhilu Wang, Jiajun Li, Xin Chen, Chao Huang, Wenchao Li, and Qi Zhu. 2024. POLAR-Express: Efficient and Precise Formal Reachability Analysis of Neural-Network Controlled Systems. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 43, 3 (2024), 994–1007. doi:10.1109/TCAD.2023.3331215
- [77] Lauren M. White, Laura Titolo, J. Tanner Slagel, and César A. Muñoz. 2024. A Temporal Differential Dynamic Logic Formal Embedding. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2024, London, UK, January 15-16, 2024*, Amin Timany, Dmitriy Traytel, Brigitte Pientka, and Sandrine Blazy (Eds.). ACM, New York, NY, USA, 162–176. doi:10.1145/3636501.3636943
- [78] Haoze Wu, Omri Isac, Aleksandar Zeljic, Teruhiro Tagomori, Matthew L. Daggitt, Wen Kokke, Idan Refaeli, Guy Amir, et al. 2024. Marabou 2.0: A Versatile Formal Analyzer of Neural Networks. In *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part II (LNCS, Vol. 14682)*, Arie Gurfinkel and Vijay Ganesh (Eds.). Springer, Cham, Switzerland, 249–264. doi:10.1007/978-3-031-65630-9_13
- [79] Kaidi Xu, Huan Zhang, Shiqi Wang, Yihan Wang, Suman Jana, Xue Lin, and Cho-Jui Hsieh. 2021. Fast and Complete: Enabling Complete Neural Network Verification with Rapid and Massively Parallel Incomplete Verifiers. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. <https://openreview.net/forum?id=nVZtXBI6LNn>
- [80] Jonathan Julián Huerta y Munive, Simon Foster, Mario Gleirscher, Georg Struth, Christian Pardillo Laursen, and Thomas Hickman. 2024. IsaVODeS: Interactive Verification of Cyber-Physical Systems at Scale. *J. Autom. Reason.* 68, 4 (2024), 21. doi:10.1007/S10817-024-09709-2
- [81] Jonathan Julián Huerta y Munive and Georg Struth. 2022. Predicate Transformer Semantics for Hybrid Systems. *J. Autom. Reason.* 66, 1 (2022), 93–139. doi:10.1007/S10817-021-09607-X
- [82] Mustafa Zeqiri, Mark Niklas Müller, Marc Fischer, and Martin T. Vechev. 2023. Efficient Certified Training and Robustness Verification of Neural ODEs. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. <https://openreview.net/forum?id=KyoVpYvWnK>
- [83] Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. 2018. Efficient Neural Network Robustness Certification with General Activation Functions. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). Curran Associates, Inc., Red Hook, NY, USA, 4944–4953. <https://proceedings.neurips.cc/paper/2018/hash/d04863f100d59b3eb688a11f95b0ae60-Abstract.html>

Received 2026-02-25; accepted 2026-05-13