University of Southampton
Faculty of Engineering and Physical Sciences
Department of Electronics and Computer Science

Joshua Smart

April 2025

Implementation of a Rocq Backend to the Vehicle Neural Network Specification Language

Supervisor: Dr. Ekaterina Komendantskaya
Second Examiner: Dr. Filip Marković

A project report submitted for the
award of MEng Software Engineering

Abstract

This project concerns the neural network (NN) specification language Vehicle. In the growing field of neural network verification, Vehicle aims to solve some of its biggest problems; Vehicle plays a role in all stages of NN development, including training, verification, and integration. System integration - cases where NNs exist as smaller components of a larger system - will be the focus of this project, as I believe this is where the most opportunity for development lies.

In this paper I extend the Vehicle language by adding support for an additional theorem prover (Rocq), widening the scope of problems Vehicle is equipped to handle and enhancing the developer experience. Furthermore, I will conduct a critical evaluation of the solution in order to verify these benefits.

This report details my investigation into the background, methods, and tools used in neural network verification; technical implementation of my solution; as well as the testing and evaluation carried out. Finally, I outline opportunities for future work and improvements based on this addition.

Statement of Originality

I have read and understood the ECS Academic Integrity information and the University's Academic Integrity Guidance for Students. I am aware that failure to act in accordance with the Regulations Governing Academic Integrity may lead to the imposition of penalties which, for the most serious cases, may include termination of programme. I consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to verify whether my work contains plagiarised material, and for quality assurance purposes.

I have acknowledged all sources, and identified any content taken from elsewhere. I have not used any resources produced by anyone else. did all the work myself, or with my allocated group, and have not helped anyone else. The material in the report is genuine, and I have included all my data/code/designs. I have not submitted any part of this work for another assessment. My work did not involve human participants, their cells or data, or animals.

Acknowledgments

I would like to thank my supervisor, Dr. Ekaterina Komendantskaya for first proposing this project to me as well as her unwavering support throughout. Her dedicated and thoughtful feedback has been invaluable to me. I also appreciate how welcome she has made me feel within the wider research community, forming connections that I will carry with me into my further academic career.

I would also like to give a special thanks to Matthew Daggitt, lead developer of Vehicle, for his insightful advice and assistance. Without his continual efforts developing Vehicle, much of this project may not have been possible. Finally, I give credit to members of the Rocq and Mathcomp communities – namely Alessandro Bruni and Reynald Affeldt – for lending me their expertise and support in these technologies.

Contents

1. Background and Literature Review

This project concerns the neural network specification language Vehicle [1], a tool developed for the training, verification, and integration of neural networks.

For most of its infancy, the field of neural networks (NNs) sat far from Verification technologies. Correctness of NNs was largely attributed to a property known as generalisability: if the network performs well on the training set as well as the test set that it has never seen before, then the network must generalise to all possible inputs with a similar degree of accuracy [2]. Additionally to this – though not commonly considered at the time – the most common use cases for NNs were ones in which the formal specification required for verification is impossible to discern. For example, developing a formal specification for recognition of handwritten digits could be at least as difficult than writing an algorithm to do so without the help of neural networks – likely more so. These two factors were what initially kept the fields apart.

1.1. Adversarial Robustness

Discovered in 2014 by Szegedy *et al.* [3] as the first example of an attack designed to undermine neural networks, an adversarial attack operates by making small changes to the input of a NN. Szegedy *et al.* showed, among others, that for the case example of a model trained on the MNIST [4] dataset of handwritten digits, they were able to construct images that were simultaneously very close to and visually hard to distinguish from an image in the dataset - but nonetheless incorrectly classified by the model [3, p. 5].

Later explicated by the verification and logic communities, the desired defensive property (coined as 'Adversarial Robustness') is defined as follows. Considering a function $f : \mathbb{R}^n \to \mathbb{R}^m$ representing a neural network. A network is *Robust* about an input $\hat{x}$ if for sufficiently small values of ε and δ [5]:

$$\forall x \in \mathbb{R}^n : |x - \hat{x}| \leq \varepsilon \Rightarrow |f(x) - f(\hat{x})| \leq \delta \tag{1}$$

This discovery cast doubt upon the assumption of generalisability within the NN community; not one but whole sets of valid data could be generated that these networks performed poorly on. This motivated the NN community to find ways to verify this kind of property, a catalyst for the field of neural network verification.

1.2. Verification of Neural Networks

Just a few years later in CAV'2017, two papers on neural network verification were published by Huang *et al.* [6] and Katz *et al.* [7]; both of the methods discussed used specialised SMT-solvers. The techniques described by Katz *et al.* would eventually become the verification tool Marabou [8] in 2019.

In the same year, the ERAN verifier was published in POPL [9] with an entirely different approach using abstract interpretation; they showed that the performance of these methods could compete with the existing SMT-solver solutions. This branched off many supporting works such as those by Muller *et al.* [10], [11].

Verifier	VNNComp 2024 Benchmark	Implementation Language
α,β-CROWN	1200.0	Python
NeuralSAT	1113.1	Python
PyRAT	1000.8	Python
Marabou	751.0	C++
nnenum	572.5	Python
NNV	530.0	MATLAB
CORA	439.5	MATLAB
NeVer2	262.3	Python

Table 1: Summary of contenders from VNNComp 2024 [12].

At this time dozens of verifiers exist, most conforming to the well established VNNLib specification, allowing compatibility and benchmarking between projects. The current state-of-the-art projects are shown in Table 1, a summary of the contenders for VNNComp 2024. These solutions have seen rapid development over the years, α,β-CROWN has been the leader of VNNComp for four years running with year-on-year improvement [12], [13]. However, while these imperative solutions have crushed previous benchmarks, there are still many challenges to overcome.

Firstly, we face the problem of discovering properties to verify. As an easily generalised example, Robustness is a popular candidate for verification, other properties are largely domain-specific (see the ACAS Xu Benchmark [7] used in VNNComp [12]).

Secondly, there is the problem of scalability. As of VNNComp 2024 [12], tools in the field could handle networks with around 70 million parameters; this is sufficient for smaller problems but for the emerging larger systems such as GPT-4 with 1.76 trillion parameters, there is still a long way to go.

Finally, neural networks exist commonly as components of a larger system. This is true even more so when considering use cases such as controllers of cyber physical systems. Integration of verification within these larger systems remains an unsolved problem [14].

1.3. Programming Language Support for NN Verification

In response to the issues discussed with the current approaches to verification, many new research-based tools have been developed - as well as existing languages extended - to support NN verification. The libraries StarChild [15] and Lazuli [15] both leverage proof capabilities in their underlying languages for NN verification, whereas CAISAR [16] takes an approach akin to Vehicle with its own DSL.

1.3.1. StarChild and Lazuli

StarChild and Lazuli [15], are two libraries that utilise refinement types for the verification of neural networks. Refinement types allow developers to specify predicates that must hold for all values of a type; for instance, a function that takes a real number and returns a real number less than 10 could be written as $f : \mathbb{R} \to \{x \in \mathbb{R} \mid x < 10\}$ [17].

```haskell
{-@ type Truthy = {v:R |  0.9 <= x && x <= 1.1} @-}
{-@ type Falsy  = {v:R | -0.1 <= x && x <= 0.1} @-}

{-@ test5 :: Truthy -> Truthy -> TRUE @-}
test5 x1 x2 = runNetwork model (2 :> [x1,x2]) == (1 :> [1])
{-@ test6 :: Falsy  -> Truthy -> TRUE @-}
test6 x1 x2 = runNetwork model (2 :> [x1,x2]) == (1 :> [0])
{-@ test7 :: Truthy -> Falsy  -> TRUE @-}
test7 x1 x2 = runNetwork model (2 :> [x1,x2]) == (1 :> [0])
{-@ test8 :: Falsy  -> Falsy  -> TRUE @-}
test8 x1 x2 = runNetwork model (2 :> [x1,x2]) == (1 :> [0])
```

Listing 1: Liquid Haskell program utilising the Lazuli library to verify the Robustness of a boolean AND network (Sourced from the Lazuli repository [18, `README.md`])

Both F* and Liquid Haskell - the languages underpinning StarChild and Lazuli respectively - use an SMT solver in their implementation of refinement types. For trivial examples this is sufficient, however neural networks can commonly contain millions of nodes and edges; as shown by Kokke *et al.*, the verification time for F*'s SMT solver increases exponentially with network size [15, p. 83]. This makes these refinement type approaches less viable for many use cases and motivates the need for more specialised solutions.

1.3.2. CAISAR

Another similar project aiming to support NN verification is CAISAR (Characterizing Artificial Intelligence Safety and Robustness) [16], currently in development at CEA List. Listing 2 shows an example specification verifying Robustness for a network trained on the MNIST dataset of handwritten digits. This specification is written in the WhyML specification language and from it CAISAR can leverage many existing 'provers' (such as Marabou [8] or PyRAT [19]) to verify the given property.

1.4. Vehicle

Taken from the project's readme:

> "Vehicle is a system for embedding logical specifications into neural networks. At its heart is the Vehicle specification language, a high-level, functional language for writing mathematically-precise specifications for your networks." [1, `README.md`]

Vehicle aims to be an all-in-one, 'batteries included' tool for NN verification. To this end, it takes a wider approach than other related tools (see Section 1.3) aiming to aid in every step of neural network development, including training, verification, and integration.

```
theory MNIST                                                    WhyML

  use ieee_float.Float64
  use caisar.types.Float64WithBounds as Feature
  use caisar.types.IntWithBounds as Label
  use caisar.model.Model

  use caisar.dataset.CSV
  use caisar.robust.ClassRobustCSV

  constant model_filename: string
  constant dataset_filename: string

  constant label_bounds: Label.bounds =
    Label.{ lower = 0; upper = 9 }

  constant feature_bounds: Feature.bounds =
    Feature.{ lower = (0.0:t); upper = (1.0:t) }

  goal robustness:
    let nn = read_model model_filename in
    let dataset = read_dataset dataset_filename in
    let eps =
    (0.0100000000000000002081668171172168513294309377670288085937500000:t) in
    robust feature_bounds label_bounds nn dataset eps

end
```

Listing 2: Example specification to verify the Robustness of a network trained to recognise the MNIST dataset of handwritten digits (Sourced from the CAISAR repository [20, `examples/mnist/mnist.why`]).

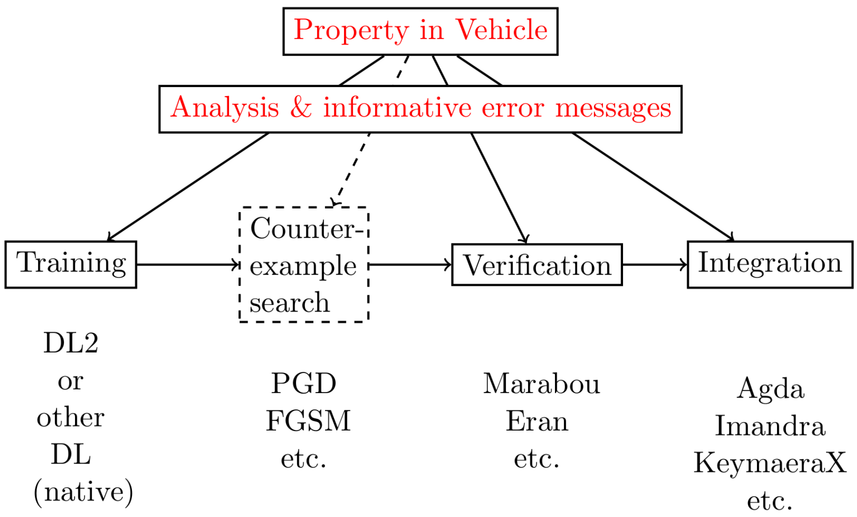

Figure 1: Backends of the Vehicle Language (sourced from Vehicle Tutorial [14, 1.2])

1.4.1. Vehicle Backends

Vehicle interfaces with an unusually broad set of tools. Figure 1 shows the various backends of the Vehicle tool[1].

The Training backend allows a Vehicle specification to be compiled into a loss function that can be used to train the network. This is achieved through the use of Differentiable Logics [21].

The Verification backend is the most commonly implemented part of NN Verification; CAISAR, Lazuli, and StarChild focus solely on this task. This allows Vehicle specifications to be compiled to various representations used by external network verifiers (currently, support is limited to the Marabou network verifier). The goal of this backend is to take the trained network and verify that it satisfies the desired property.

Finally, the Integration backend forms the most novel addition to the space of NN verification – and the focus of this project. This allows specifications to be compiled to languages used for system verification (current support is limited to the proof assistant Agda). The goal of this backend is to use the verified statement from the Verification backend to prove desired properties of a larger system [22].

1.4.2. Vehicle DSL

The Vehicle DSL is a statically typed, functional programming language in which users write network properties. Listing 3 is taken from the examples in the Vehicle repository and shows the specification of a car controller neural network.

Vehicle implements many types common in programming languages, such as `Bool`, `Int`, and `List`; as well as others specific to this use case, e.g. `Index` and `Tensor`. See Appendix B.1 for a full breakdown of types.

The language also contains a collection of directives for declaring connections with external tools and data. See Appendix B.2 for full descriptions of the available directives.

Vehicle also contains both universal and existential quantifiers to aid in writing properties; these are accessed by the `forall` and `exists` keywords.

[1]Note that the 'Counter-example search' backend is currently unimplemented in Vehicle and represents an opportunity for future development.

```
type InputVector = Tensor Rat [2]

currentSensor  = 0
previousSensor = 1

type OutputVector = Tensor Rat [1]

velocity = 0

@network
controller : InputVector -> OutputVector

normalise : InputVector -> InputVector
normalise x = foreach i . (x ! i + 4.0) / 8.0

safeInput : InputVector -> Bool
safeInput x = forall i . -3.25 <= x ! i <= 3.25

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

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

Listing 3: Example specification in the Vehicle DSL (sourced from Vehicle examples [1, `examples/windController/windController.vcl`])

1.5. Comparison of Proof Assistants

Proof assistants make up the underlying mechanism of the Integration backend discussed in Section 1.4.1, providing an interactive system for writing and checking proofs. In the context of Vehicle, they can be used to describe a larger system that contains a NN. Then using the externally proved property from the specification, properties can be proven about the system as a whole. This is instrumental in verifying safety constraints for mission-critical cyber physical systems.

In general, proof assistants consist of interactive programs to aid the user in obtaining verified propositions. This is achieved in two ways; automatically via a theorem prover, or the proof can be written by the user and checked by the program [23, 5]. The first program that could be considered a proof assistant was Automath in 1967, which provided a proof script language as well as a proof checker to verify them; this language leveraged the Curry-Howard correspondence that is seen ubiquitously among modern proof assistants. This inspired many later projects such as NuPrl in 1984; and Mizar in 1973, the longest continuously running proof assistant project [24].

I will discuss the benefits and drawbacks of several proof assistants, including Agda [25] (the ITP currently supported by Vehicle), Rocq[2], and Idris[3]. My aim is to decide which proof assistant would be most beneficial to add to the Vehicle language.

1.5.1. Language Features

While Agda and Idris both reflect the syntax of Haskell, Rocq borrows its syntax from OCaml. From a user standpoint, I don't believe this makes a significant difference as all three support patterns and usages common among most functional languages.

1.5.2. Proof Assistant Features

While Agda and Idris are mostly comparable in this category, relying on constructive proof semantics, Agda provides a more mature and capable experience than Idris. Rocq's user experience differs greatly, promoting backwards reasoning through its large tactic library. Implementing a language with an alternative interface such as Rocq's could be beneficial to the breadth of use cases that Vehicle is equipped to handle, as well as encouraging a larger user-base.

1.5.3. Libraries

A notable weakness of Agda is its lack of external libraries for domains such as calculus and analysis, common topics for applications of NNs. This severely limits the number of viable use cases, as the user would need to reimplement much of the domain-specific logic; Idris shares this problem but to a lesser degree. The foremost of this category is Rocq, with a rich ecosystem of libraries for most problem domains. This support could be great for expanding the number of problem domains that Vehicle can cover.

1.5.4. Tooling

In this category both Agda and Rocq excel, both with well maintained VSCode plugins and Emacs modes. Idris does not share this support; the Idris Emacs mode has received many recent updates but is still behind the functionality of Agda. Developer experience is a key metric for this comparison, and Rocq's on-par support with Agda makes it a great contender for inclusion within Vehicle.

[2]In late 2023, Rocq was renamed from its original name Coq. This went into effect largely around early 2025, midway through this project. To avoid confusion it will be referred to as Rocq throughout this report.

[3]Note that Idris2, the successor to Idris, is currently in development. However, currently lacks many core features of proof assistants. For this reason I will use Idris in this comparison.

2. Project Goals and Methodology

The goal of this project is to extend the capability of the Vehicle language by implementing support for additional ITP backends. This is motivated by Agda's relative lack of support for fields such as Calculus and Analysis, two topics that are common in cyber physical systems, as well as its high barrier for entry to new users. This project's scope is limited to integration of a Rocq backend. Rocq was chosen as it should solve many of the problems experienced with Agda; Rocq has wide support for Calculus and Analysis through the `mathcomp`[26] library; and Rocq's tactic proof system should reduce the complexity and difficulty of learning for new users (see Section 1.5).

2.1. Backend Implementation

The user interface for the Rocq backend should parallel the Agda backend with the following usage, which should be simple to implement by analogy to the existing Agda integration:

```shell
vehicle export --target Rocq --cache cache-path --output Spec.v
```

The most challenging aspect of this project will be the compilation from the Vehicle specification language into Rocq; there are a few design choices to be made here. Firstly, the decision to use rational numbers or real numbers inside Rocq. Secondly, I will need to explore how structures such as tensors and predicates can be translated between Vehicle and Rocq. And finally, I will need to investigate how to express unproven theorems and undefined functions within Rocq to represent the externally proved property and external network respectively.

The project's development will also require a suite of tests to give confidence in the correctness of the implementation. Taking inspiration from the methods used in the existing Vehicle project, I will create a suite of Golden tests to validate my implementation.

2.2. Evaluation

Firstly, I will verify that the test cases mentioned in Section 2.1 are all successful; this will be a good indication that my solution is technically functional.

Then, in order to evaluate the usefulness of my solution to an end user, I will first use it to verify the `windController` specification from the Vehicle examples [1, `examples/windController`]. I will add this example to the Vehicle repository to aid future users. Secondly, I will attempt to show a more complex example which integrates advanced topics such as calculus that are unavailable in Agda.

3. Technical Implementation

In this section I will outline the structure and implementation of the Vehicle Rocq backend, including notable design decisions and technical details.

This project develops on top of the existing Vehicle codebase; after conversation with the Vehicle development team, it became apparent that there was a major refactor in progress at the time this project started. A new branch `tensor-refactor` was in development that would improve the ergonomics and simplify the internal structure of Vehicle by altering the way tensor data structures were stored and handled. At their recommendation, I have chosen to implement this new feature on top of the `tensor-refactor` branch as it is due to be merged soon, and will include significant changes to the implementation details of this project.

Vehicle is developed in Haskell, a popular functional programming language well suited to applications such as compilers. In this project I will design and implement the `Vehicle.Backend.Rocq` module, the functionality of which lies primarily in the following function definition:

```haskell
compileProgToRocq :: (MonadCompile m) => Prog DecidabilityBuiltin
-> RocqOptions -> m (Doc a)
```

Haskell

`Prog DecidabilityBuiltin` defines the type of a Vehicle AST where the expressions have been checked for decidability (the distinction between `bool` and `Prop` in Rocq). Decidability checking is a new feature in the `tensor-refactor` branch, replacing the old heuristic method. And the return type `Doc` is part of the `prettyprinter` package that allows for clean output of compiled code.

3.1. Dependency Enumeration

Rocq implements a mature and full-featured module system, allowing users to structure large projects and construct clean abstractions. As a core function of the backend, the program will need to track the required dependencies of the program in order to generate the necessary preamble; for instance, the Rocq expression `(1%N :: nil)` for constructing a list requires the `mathcomp.ssreflect.seq` and `mathcomp.ssreflect.ssrnat` imports.

The three statements that must be considered are: `Require Import <name>`, used to load a module and bring its definitions into scope; `Import <name>`, used to bring the definitions of a module that is already loaded into scope; and `Open Scope <name>`, used to add the scope's notations onto the stack, allowing them to be parsed.

Listing 5 shows how these structures are represented in Haskell. The compiler uses the `prettyprinter`, package to track and collate dependencies. During compilation, whenever an expression is encountered that requires a dependency, the returned text is 'annotated' with its dependency list. Using this system, after compilation these dependencies can be collated and appended to the beginning of the text.

```haskell
data Dependency
  = RequireImport Library
  | Import Module
  | Open Scope
  deriving (Eq, Ord)
…
data Library
  = MathcompSsreflectSsrbool
  | MathcompAlgebraSsralg
…
  | VehicleTensor
  | VehicleStd
  deriving (Eq, Ord)
…
data Module
  = DefaultTupleProdOrder
  deriving (Eq, Ord)
…
data Scope
  = RingScope
  | TensorScope
  | OrderScope
  deriving (Eq, Ord)
```

Listing 5: Rocq dependency representation [1, `vehicle/src/Vehicle/Backend/Rocq/Compile.hs`]

3.2. Promoting Rationals to Reals

In past, mismatches of numerical types have caused issue for NN verifiers [27]; a known limitation of the Vehicle language is its internal reliance on rational numbers. This is inconsequential when compiling to Agda as the language has little support for reals. However, as discussed in Section 1.5, Rocq has much wider support for real numbers in the Mathcomp library. This begs the question, which numerical system should the Rocq backend use? In the following section I will evaluate the benefits and drawbacks of each, motivating the design of the project. To facilitate this investigation, I will use the car controller example shown in Section 1.4, attempting to manually translate the Vehicle specification (Listing 3) into Rocq using each of the numerical systems, as well as creating the surrounding system verification.

Documented in Appendix C.1 and Appendix C.2 are the handwritten Rocq files containing the network property specified in Vehicle and the requisite system proof. Note the use of `Parameter` and `Axiom` to denote both the abstract `controller` function representing the neural network, and the externally proved `safety` property respectively. Both of these files utilise the QArith package for rational numbers; this caused some issues when it came to proving properties of the system. The main issue was due to rational numbers not having unique definitions, a fact that does not play well with Rocq's reliance on syntactic equality. This made filling out the proofs exceedingly difficult and, at this point, I decided to attempt it using reals.

Converting the example to utilise the Mathcomp library was fairly simple, see Appendix C.3 for the example file. This exercise gave me key insights into the problem; Mathcomp is a well-maintained library with a helpful and engaging community - a few major issues with my approach were aided by members of that community. This benefit could extend to users of this project. Furthermore, Mathcomp's range of tactics such as `ring` and `lra` have been and will continue to be instrumental in proving the wider system properties.

On balance, I believe that leveraging the powerful Mathcomp library to implement the backend using real numbers will not only widen the potential problem scope of Vehicle (real numbers being the more natural representation of this problem), but also provide a mature and capable developer experience.

Vehicle

```
number : Rat
number = 5
```

Rocq

```
1 Require Import mathcomp.algebra.ssralg.
2 Require Import mathcomp.reals.reals.
3 Open Scope ring_scope.
4 Parameter R : realType.
5
6 Definition number : R := 5.
```

Listing 6: Translation of rational numbers.

Listing 6 shows the translation between a statement of type `Rat` in Vehicle into Rocq. Note the required imports as well as the `Parameter` statement on line 4. This is used to declare an opaque instance of the `realType` interface, which is required once at the top of each specification that requires real numbers; the `Parameter R : realType.` statement is implemented as a special case to the preamble logic that is only included if the `mathcomp.reals.reals` dependency is present.

3.3. Mathcomp Integration

The Mathematical Components Library (Mathcomp) is a set of canonical formalisations for a wide range of mathematical theories and structures such as lists, prime numbers, and finite graphs. These libraries have been used exensively in formal proof, notably in a proof of the Four Colour Theorem [28]. Mathcomp leverages the SSReflect proof language, an alternative proof syntax that is well supported by both the tooling and the community, vastly extending its capability.

Many types and concepts from Vehicle map directly onto constructs in Mathcomp; for example, Vehicle's `Index n` type which can be interpreted as $\{m \in \mathbb{N} \mid m < n\}$ is represented in Rocq using Mathcomp's `fintype` package, with `ordinal n` as the equivalent notation.

```
i : Index 5
i = 3
```

```
Require Import mathcomp.algebra.ssralg.
Require Import mathcomp.ssreflect.fintype.
Require Import mathcomp.algebra.zmodp.

Definition i : ordinal 5 = 3
```

Listing 7: Translation of Index types.

Definitions lifted to Mathcomp include but are not limited to: index types; real numbers (shown in Listing 6) and associated arithmetic; lists and list operations; ordering; equality; and boolean operations. This comprehensive integration allows users to leverage the mature formalisations around these structures.

To mention some edge cases within this integration, comparison on index types are excepted from the usual definitions found in `mathcomp.ssreflect.order`. This is because Vehicle supports comparison between indexes of differing orders, a semantics not compatible with the `eqType` and `orderType` interfaces as they require both operands to be of convertable types. The solution implemented by the backend is given below, in which the ':>' notation is used to cast both operands to the specified type before comparison, allowing the behaviour shown in Vehicle.

```
a : Index 1
a = 0
b : Index 2
b = 1

ltIndex : Bool
ltIndex = a < b
```

```
(* Preamble ommitted *)

Definition a : ordinal 1%N := 0.
Definition b : ordinal 2%N := 1.

Axiom ltIndex : a < b :> nat.
```

3.4. Vehicle-Rocq Companion Library

Vehicle maintains its own standard library of commonly used functions, for instance the function `appendList`:

```
appendList : List A -> List A -> List A
appendList xs ys = fold (\x y -> x :: y) ys xs
```

This necessitates the creation of a Rocq-based companion library to be shipped to users alongside Vehicle. The so-called `vehicle-rocq` library holds translations for all of Vehicle's standard library definitions as well as additional constructs present in Vehicle, which cannot be sourced from external libraries such as Mathcomp. It is a strong design principal that the API surface of this library be kept as small as possible, preferring to use existing definitions. This ensures that generated specifications are as similar as possible to one a user may write by hand; this also has additional benefits to the maintainability of the Vehicle project as a whole.

However, one such missing construct is the `Tensor` type; as of the writing of this report, Mathcomp does not yet contain a formalisation for the tensor (or multilinear map) data structure. This motivated my investigation and implementation of a custom tensor structure for Vehicle, which is documented in Section 3.5.

```
vehicle-rocq
├── _CoqProject
├── Makefile
├── std.v
├── tensor.v
└── vehicle-rocq.opam
```

Listing 8: Vehicle-Rocq library structure [1, `vehicle-rocq/`]

The library consists of two proof scripts, shown in Listing 8. These are distributed alongside both Rocq and Opam package definitions, making this library as easy as possible for users to depend on it in their own projects.

3.5. Tensor Representation

In the context of computer science and machine learning, a *tensor* represents a generalization of multidimensional arrays. An N-dimensional tensor is a structure that requires N indices to access; a 1-dimensional tensor is a vector, a 2-dimensional tensor is a matrix, and tensors with dimension 3 or higher are known as higher-dimensional tensors [29].

The tensor structure is isomorphic to another structure in linear algebra called a *multilinear map*. Formally, a multilinear map is a function $f : V_1 \times \ldots \times V_n \to W$ where all V_n and W are vector spaces; f must be linear with respect to each variable, meaning that for all i, if all variables but v_i are constants, then $f(v_1, \ldots, v_i, \ldots, v_n)$ is a linear function [30].

As mentioned in Section 3.4, this project necessitates the implementation of a custom tensor data structure. I decided to implement it using a nested-tuple approach (a tuple describes a list with a fixed, known length), the definition is as follows:

```rocq
Definition tensor (A : Type) : seq nat -> Type :=
    foldr tuple_of A.
```

Listing 9: Tensor type definition [1, `vehicle-rocq/tensor.v:8`]

To disambiguate this definition, `tensor` is defined as a higher-order type (a type parameterized over other types or values) where `A` represents the kind of value the tensor contains, and the `seq nat` argument (a sequence of natural numbers) representing its dimensions. Therefore the type of 6 by 7 matricies of real numbers can be described with `tensor R [:: 6; 7]`. This definition has a few nice properties for operations over this data structure; firstly consider the `stack` operation with signature:

```rocq
Definition stack {A d} : d.-tuple (tensor A nil) -> tensor A (d ::
nil) := id.
```

This function is trivially easy to implement as the two types are definitionally equal. In fact, under Rocq's type system, it can be shown that `stack` is equivalent to `id`, the identity function. Secondly, it follows naturally that a 0-dimensional tensor (or nil-tensor) should represnt only a single value of its contained type, that is to say: `tensor T [] = T` which is trivially true with this definition. However, in constrast to the properties stated above, the remaining operators are not so cleanly defined; the `zipWith` operation takes as input 2 tensors of equal dimension and constructs a new tensor by applying a given function pointwise.

```
Fixpoint zip {A B ds} : tensor A ds -> tensor B ds -> tensor (A * B)
ds :=
    match ds with
    | [::] => pair
    | d :: ds => fun xs ys => [tuple zip (tnth xs i) (tnth ys i) | i < d]
    end.

Definition zipWith {A B C ds} (f : A -> B -> C) (xs : tensor A ds) (ys :
tensor B ds) : tensor C ds :=
    map (uncurry f) (zip xs ys).
```

The reliance on recursion and heavy use of constructs in the underlying tuple library make this definition difficult to reason with, requiring multiple levels of structural induction. It is suggested that future research may include a Mathcomp-native tensor formalisation that could replace this definition. The benefits this would bring are discussed further in Section 6.1. At this stage users are recommended not to rely on tensor arithmentic for logical property statements, and insead to rewrite their statements using universal quantification, an example of this is given in Section 4.4.

3.6. Canonical Function Definitions

One of the most distinct ways in which Rocq's syntax differs from Vehicle's occurs with top-level declarations. Vehicle takes an approach similar to its parent language Haskell, with separate type and body declarations. Whereas, in Rocq, arguments and return type are declared separately within the same statement.

```
double : Nat -> Nat
double n = n * 2
```

```
Definition double (n : N) : N
  := n * 2.
```

Listing 10: Translation of top-level declaration

Listing 10 shows the desired canonical translation. However this comes with some additional difficulties; the return type is not simply available from the Vehicle AST, instead the declaration is annotated with its full function type (`Nat -> Nat` in the above example). This necessitated the implementation of a return type resolution algorithm, further complicated by Vehicle's support for dependent typing allowing signatures such as `dependently_typed : forallT (t : Type) . t -> t`.

To conclude, this functionality constituted one of the more involved parts of this project, requiring investigation into the intricacies of both Vehicle and Rocq's type systems. Nonetheless, I believe it to be an important feature that greatly improves readability of the generated scripts compared to previous workarounds I had implemented. The source code for this algorithm can be found here.

3.7. Partially Applied Infix Notations

Rocq boasts a powerful notations system, allowing users to influence the way expressions are parsed. This is in fact the same mechanism used by the language itself to handle infix operators, an example of this is given below:

```
Notation "A /\ B" := (and A B).
```

However, this poses some problems for the Rocq backend. Consider the AST of a Vehicle expression with the following structure:

```
Application (LessThan) [(Rat 5)]
```

Note that the less-than function has been partially applied. Naively this would be compiled to `"5 <"` or throw a compile-time error regarding an incorrect number of arguments. However, this kind of partial application is valid in functional languages such as Haskell or Vehicle.

This expression would compile to `"5 <_"` in Agda, in which the underscore replaces the ommitted argument and evaluating to the expected function. There is no analogous feature in Rocq with its more generalised notation system. To solve this issue the backend uses a fallback system where expressions that would be processed as infix first check if the correct number of arguments has been supplied, otherwise it evaluates to an application in the more functional style. For instance the above expression compiles to `"le 5"` (where `le` is the function underlying the < notation). This solution matches the expressiveness of the Vehicle language while preserving the neater infix syntaxes where possible.

3.8. Reflections From Bool to Prop

One of the welcome properties of the Rocq proof language is the ability to implicitly reflect from `bool` to `Prop` (from decidable boolean expressions to undecidable expressions at type-level). This is ubiquitous in Mathcomp's implementation where most computation is done in `bool` and reflected to type-level when necessary.

This feature is facilitated by the `is_true` coercion defined in Rocq's standard library, which means that any boolean expression can be treated as a `Prop` with no extra syntax. This has user-facing benefits in the Rocq backend; all predicate operations such as equality and comparison can be implemented once in `bool` and used in both decidable and undecidable expressions.

```
f : Rat -> Rat -> Bool
f a b = if a < b then True
  else a == b
```

```
Definition f (a : R) (b : R) : Prop
  := if a < b then True else a < b.
```

Listing 11: Translation of an expression with both decidable and undecidable comparison

Note how in Listing 11, the < operator is used both in a decidable context (the condition of an if statement) and an undecidable one (the result of a function returning `Prop`). This represents a point of simplicity compared to the Agda backend, wherein each operator must have decidable and undecidable variants.

3.9. Documentation Updates

Alongside the other changes made in this project, I have also made updates to Vehicle's documentation. This includes Vehicle's language documentation, as well as the README and examples documentation. These sources outline the main functionality of the Rocq backend as well as discussing some of its strengths and limitations. An excerpt of these changes can be found in Appendix F. This will aid users in both the useages and more nuanced behaviours of the Rocq backend.

3.10. Changes to Existing Code

This project did not require significant changes to existing code. The existence of the Agda ITP backend meant that much of the surrounding infrastructure was already implemented. However, I will now go on to discuss the exceptions to this rule.

3.10.1. Type/Prop Distinction

One large distinction between the type systems of Agda and Rocq lies in their handling of sorts (types of types). Conventionally, Agda labels all type-level expressions – including propositions and types themselves – as sort `Set`. Whereas, Rocq defines a more complex hierarchy of sorts; for this application we must consider the `Type` and `Prop` sorts, representing types and propositions respectively. The compilation difference this poses is illustrated below:

```agda
Vec2 : Set
Vec2 = Tensor ℚ (2 ∷ [])

Positive : Vec2 → Set
Positive x = Fin.All
  (λ i → x ! i > 0)
```

```coq
Definition vec2 : Type
  := tensor R (2 :: nil).

Definition positive (x : vec2) : Prop
  := forall i, tnth x i < 0.
```

Observe how the expressions both result in the sort `Set` in Agda, but are split between `Type` and `Prop` in Rocq. This required some changes to the mechanism that produces the intermediate representation used for compilation, as the necessary information to make this distinction was being discarded. I would like to thank Matthew Daggitt for assisting me in making this change.

3.10.2. Line vs Block Comments

Finally, the last change made was to the logic surrounding comment formatting. I added additional functionality to facilitate block comments as well as line comments, as Rocq only supports block-style commenting. This was a minor but crucial change in which I was diligent to not introduce regressions into the codebase.

4. Testing

4.1. Testing Methodology

In order to validate the efficacy of the solution, I have implemented a suite of golden test cases. Golden testing (or Characterization testing) is used to verify the invariance of some behaviour [31]; for example, for some algorithm $f : \mathbb{R} \to \mathbb{R}$ if we have that $f(2) = 4$, where 4 is the known and verified correct value, then we can implement this as a golden test. In this way the system is protected against future unintentional modifications to this behaviour.

Vehicle's golden tests are implemented using the tasty-golden package as well as the custom test runner tasty-golden-executable. Each test case is defined in a `test.json` file, an example of which is given in Listing 12.

```json
1  [
…                                   …
13   {
14     "name": "Rocq",
15     "run": "vehicle compile -s spec.vcl -t Rocq -o Rocq.v -c
       Marabou.queries",
16     "needs": ["spec.vcl"],
17     "produces": ["Rocq.v"]
18   },
19   {
20     "name": "RocqVerify",
21     "run": "vehicle compile -s spec.vcl -t Rocq -o Rocq.v -c
       Marabou.queries && coqc -vok Rocq.v -w none",
22     "needs": ["spec.vcl"],
23     "produces": ["Rocq.v"],
24     "external": ["coqc"],
25     "ignore": {"files": [".Rocq.aux", "Rocq.glob", "Rocq.vok"]}
26   },
…                                   …
46 ]
```

Listing 12: Example `test.json` spec [1, `vehicle/tests/golden/compile/andGate/test.json`]

Lines 15 and 21 show the commands executed in these tests. Note that the `RocqVerify` test also runs the output through the Rocq compiler using the `-vok` type-checking flag as an additional validation step.

4.2. Continuous Integration

I have also implemented a CI pipeline using GitHub Actions, this operates by creating a consistent and reproducible environment using docker in which to run the tests. In order to run the tests, the following software must be installed:

1. **Haskell**, to compile Vehicle.

2. **Ocaml** and **Opam**, to install Rocq and required dependencies.
3. **Rocq**, to type check the generated specifications, as mentioned in Section 4.1.
4. **Mathcomp**, as a dependency to all compiled specifications, as well as the Vehicle-Rocq companion library.
5. **Vehicle-Rocq** library, as a dependency to compiled specifications.

This pipeline will verify that future changes to Vehicle do not introduce regressions into the codebase. See Appendix D for the full workflow configuration.

4.3. Test Report

Given below in Listing 13 is the test report for all cases concerning the Rocq backend, the `<test>.Rocq` cases validate that the compiled output matches the given and independently verified 'golden' files (more detail on golden testing can be found in Section 4.1). For each of these cases there is an accompanying `<test>.RocqVerify` case which includes the additional type-checking step described above.

```
Compiler
  compile
    simple-quantifierIn
      Rocq:       OK (0.39s)
      RocqVerify: OK (4.60s)
    monotonicity
      Rocq:       OK (0.20s)
      RocqVerify: OK (5.78s)
    simple-index
      Rocq:       OK (0.24s)
      RocqVerify: OK (4.60s)
    simple-if
      Rocq:       OK (0.41s)
      RocqVerify: OK (5.90s)
    dogsHierarchy
      Rocq:       OK (0.88s)
      RocqVerify: OK (6.36s)
    simple-quantifier
      Rocq:       OK (0.48s)
      RocqVerify: OK (6.05s)
    simple-tensor
      Rocq:       OK (0.42s)
      RocqVerify: OK (5.97s)
    increasing
      Rocq:       OK (0.13s)
      RocqVerify: OK (5.77s)
    autoencoderError
      Rocq:       OK (0.31s)
      RocqVerify: OK (6.00s)
    mnist-robustness
      Rocq:       FAIL (0.28s)
      RocqVerify: FAIL (0.26s)
    simple-let
      Rocq:       OK (0.39s)
      RocqVerify: OK (5.97s)
    simple-vector
      Rocq:       FAIL (0.16s)
      RocqVerify: FAIL (0.17s)
    windController
      Rocq:       OK (0.46s)
      RocqVerify: OK (6.02s)
    simple-constantInput
      Rocq:       OK (0.15s)
      RocqVerify: OK (5.71s)
    acasXu
      Rocq:       OK (5.13s)
      RocqVerify: SKIP (disabled)
    andGate
      Rocq:       OK (0.83s)
      RocqVerify: OK (5.72s)
    simple-arithmetic
      Rocq:       OK (0.23s)
      RocqVerify: OK (5.15s)
    reachability
      Rocq:       OK (0.17s)
      RocqVerify: OK (3.25s)
    simple-pruneDecls
      Rocq:       OK (0.22s)
      RocqVerify: OK (3.34s)

5 out of 38 tests failed (15.51s)
```

Listing 13: Rocq backend test report

4.3.1. Failing Tests

However, Listing 13 also shows a number of failing test cases. Each of which I will now explain in further detail.

Firstly, regarding the `mnist-robustness` test cases, these cases fail due to an upstream issue in the `tensor-refactor` branch that has not yet been fixed. However, this test does not introduce any features or syntax not expressed by other cases, so I am confident that when the underlying issue is fixed that `mnist-robustness` will pass without error.

Secondly, within the ongoing process of refactoring Vehicle's tensor representation, the developers will be replacing the intrinsic `Vector` type with the more general `Tensor` type, so the `simple-vector` case will likely be removed in the near future in favour of `simple-tensor`.

Finally, the `acasXu` test cases are an anomaly; although the correct specification is generated, it cannot be type-checked by the Rocq compiler. The error message generated is `'stack overflow'` pointing to the following line:

```
Definition pi : tensor R nil := 392699 / 125000 : R.
```

Rocq

The cause of this error lies within the numeric literals themselves; internally, the `392699` literal is first represented as a natural number – its default interpretation – and then coerced into an element of the `realType` instance. However, Rocq natural numbers are defined using Peano axioms meaning that the literal `392699` attemps to expand to almost four-hundred-thousand applications of the succsessor function, hence the stack-overflow error. This is unfortunately a limitation of Rocq's standard library definition for natural numbers; this could be remedied by integrating with other libraries such as bignums or alternatively, if in future Mathcomp were to include a native notation for real numbers, then this test would be successful without the inclusion of additonal libraries.

4.4. Autoencoder Example

In the following sections, I will give usage examples of my work, describing an end-to-end process of problem definition to formal proof. The first example, a novel application of Vehicle devised for this project, concerns an autoencoder. An autoencoder refers to a specific kind of neural network used to find efficient encodings of data, it is comprised of 2 underlying networks that are trained in tandem. The *encoder* takes the input data and transforms it into the underlying representation, and the *decoder* takes data in the form of the underlying representation and transforms it back into the original input. The definining property of this system is that the *encoder* and *decoder* should be approximate inverses of one another, formally:

Let X be the space of the input data, and Z be the space of encoded data. We define two networks $E_\varphi : X \to Z$ and $D_\theta : Z \to X$ where φ and θ represent the parameters of the respective networks. We can then state the desired property as:

$$\forall x, \left| D_\theta\left(E_\varphi(x)\right) - x \right| < \varepsilon \tag{2}$$

Where ε denotes the maximum "reconstruction error". The two primary uses of autoencoders are dimensionality reduction and information retrieval; dimensionality reduction is used in applications such as compression where $|X| \gg |Z|$ allowing for efficient reduction in data size. Information retrieval applications can make use of specially trained binary encodings in order to group related data [32].

```
1  @network
2  encode : Tensor Rat [5] -> Tensor Rat [2]
3
4  @network
5  decode : Tensor Rat [2] -> Tensor Rat [5]
6
7  epsilon : Tensor Rat [5]
8  epsilon = foreach i . 0.1
9
10 @property
11 identity : Bool
12 identity = forall i x . x ! i - epsilon ! i <= decode (encode x) ! i <= x ! i + epsilon ! i
```

Listing 14: Autoencoder Vehicle specification, [1, `examples/autoencoderError/spec.vcl`]

Listing 14 shows the specification for this example. Lines 2 and 5 give the network declarations relating to E_φ and D_θ, and line 12 denotes Equation 2 described using Vehicle's specification language.

Alternatively, the `identity` property could more consisely be written as:

```
identity = forall x . x - epsilon <= decode (encode x) <= x + epsilon
```

However, as mentioned in Section 3.5, the logical property utilising universal quantification is much easier to reason about compared to properties over tensor arithmetic.

Next, utilsing the Vehicle tool, we can first train and verify the networks against this property; details of the training and verification backends can be found in Section 1.4.1. Once we have successfully trained and verified our models, we can then put the contribution of this project to use in compiling the specification from the Vehicle langauge into Rocq.

Rocq

```
…
Parameter encode : tensor R (5%N :: nil) -> tensor R (2%N :: nil).

Parameter decode : tensor R (2%N :: nil) -> tensor R (5%N :: nil).

Definition epsilon : tensor R (5%N :: nil) := foreach (fun i => 1 / 10 : R).

Axiom identity : forallIndex (fun i => forall x, ((tnth x i -%t tnth epsilon i) <= tnth (decode (encode x)) i) /\ (tnth (decode (encode x)) i <= (tnth x i +%t tnth epsilon i))).
```

Listing 15: Abridged compiled Autoencoder specification

Listing 15 shows an abridged version of the compiled specification in Rocq; see Appendix E.1 for the full listing. Note that although the `identity` definition appears to use the tensor arithmetic operators `-%t` and `+%t`, these are operations over tensors of type `tensor R []`, so reduce trivially to operations over real numbers during simplification. Finally, now that we have a compiled and verified specification, we can use this for further proof, the example of which is shown in Listing 16; note the use of the `identity` property provided by the Vehicle specification.

Rocq

```
…
Require Import autoencoderErrorSpec.
…
Lemma closure : forall i x, let
    y := decode (encode x) in
    tnth minValue i <= tnth x i <= tnth maxValue i
    -> tnth minValue i - tnth epsilon i <= tnth y i <= tnth maxValue i + tnth epsilon i.
Proof.
    move=> i x y /andP [mx Mx]. have [Im IM] := identity i x. apply /andP; split; rewrite /y.
    - apply /le_trans; last by apply Im.
      apply lerB. by apply mx. by [].
    - apply /le_trans; first by apply IM.
      apply lerD. by apply Mx. by [].
Qed.
```

Listing 16: Autoencoder closure proof [1, `examples/autoencoderError/rocqProof/Proof.v`]

In this section, we have seen how a user can take a well defined problem; write out a Vehicle specification for the desired network properties; train and verify a network; compile the specification into the Rocq language; and finally utilise the verified specification to prove properties about a larger system.

4.5. Wind Controller Example

To summarise my second contribution to the Vehicle examples, we will use the scenario of a system entailing a car on a road that is controlled by a neural network. This example is widely cited in the Vehicle literature and I will restate it here while altering the methodology to make use of the new Rocq backend. Note that the training and verification steps have been ommited from this example for brevity.

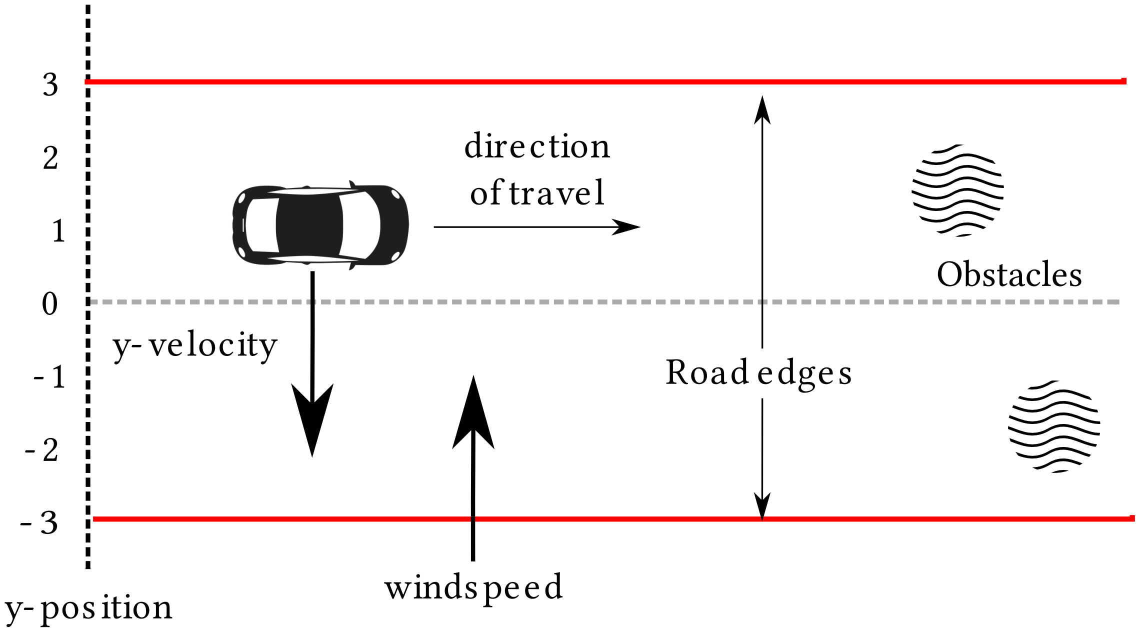

Figure 2: Wind controller system diagram (sourced from [22, 2.1])

To summarise Figure 2, an autonomous vehicle is travelling parallel to the x-axis along a road of width 6. The car experiences a cross-wind perpendicular to its direction of travel, and can respond by altering its own y-velocity. The car is equiped with a sensor that can imperfectly read its position on the y-axis. As inputs the car's controller takes in both the current and previous sensor readings. We then set out to prove the following theorem:

> **Theorem 1.** Assuming that the wind-speed can shift by no more than 1 per unit time and that the sensor is never off by more than 0.25 then the car will never leave the road.
>
> — [22, 2.1]

Clearly, this theorem is not simply a property of the network, but intead a property of the system as a whole, a textbook application for Vehicle's ITP backends. It can be shown that the necessary property of the network is as follows (the details of which can be found in the original paper [22, 2.1]):

$$\forall xy \to |x| \leq 3.25 \to |x| \leq 3.25 \to |\text{controller } x\ y + 2 * x - y| < 1.25 \qquad (3)$$

Equation 3 can then be formalised into a Vehicle specification, shown earlier in Listing 3. After which, the Rocq backend is used to compile the specification into Rocq; the compiled specification is given in Appendix C.3. From this we can construct a formal description of the system, starting with records representing the state of the system and the sensor's observations. This mirrors the process taken using the Agda backend.

Rocq

```
…
Record State :=
    { windSpeed : R
    ; position : R
    ; velocity : R
    ; sensor : R
    }.

Record Observation :=
    { windShift : R
    ; sensorError : R
    }.
…
```

Assuming the existence of a function $controller : R \to R \to R$ which takes as arguments the previous and current observations (this function is provided by the compiled specification), we can then go on to define the system's iteration:

Rocq

```
…
Definition initialState : State :=
    {| windSpeed := 0
    ; position := 0
    ; velocity := 0
    ; sensor := 0
    |}.
…
Definition nextState (o : Observation) (s : State) : State :=
    let newWindSpeed := s.(windSpeed) + o.(windShift) in
    let newPosition := s.(position) + s.(velocity) + newWindSpeed in
    let newSensor := newPosition + o.(sensorError) in
    let newVelocity := s.(velocity) + controller newSensor s.(sensor) in
    {| windSpeed := newWindSpeed
    ; position := newPosition
    ; velocity := newVelocity
    ; sensor := newSensor
    |}.
…
```

Pairing this and some other natural definitions, we can formalise Theorem 1.

```Rocq
…
Definition finalState (xs : seq Observation) : State :=
    foldr nextState initialState xs.
…
Lemma finalState_onRoad :
    forall xs, (forall x, Coq.Lists.List.In x xs -> validObservation x) ->
    onRoad(finalState xs).
…
```

The full 238-line definition and proof of this property can be found in Appendix C.4. This example has shown how the Rocq backend can be used to verify more complex properties where the NN makes up only a component of the wider system.

5. Critical Evaluation

I believe this project was a success. I have acomplished the goal set out in my project brief (given in full in Appendix A): "to add support for other ITPs as a backend to the Vehicle language". At this time, the new Rocq backend is functional, extending the capability and accessibility of the Vehicle tool. From the goals set out in Section 2: I have implemented a new backend based on the Rocq proof language; this backend integrates tighly with the popular and well-maintained Mathcomp library; the backend integrates seamlessly with Vehicle's existing command-line-interface; finally, I have implemented both a comprehensive suite of tests as well as provided functional examples to verify the efficacy of this solution.

However, there was one goal not met within this project. To provide an additional usage example that demonstrated the advanced representation and proof capabilites of Mathcomp within the Vehicle pipeline. This was primarily due to both time and prioritisation constraints, more details are given in Section 5.2. Nonetheless, this ommission does not invalidate the success of this project, I have yet shown in the following section the largely comparable efficacy between this project and its Agda counterpart.

5.1. Comparison to Existing Agda Backend

I believe that the Rocq backend implemented in this report is as capable as the Agda backend. As shown by the test report in Section 4.3, all specifications compilable to Agda can also be compiled to Rocq. Additionally, outlined in Section 4.5, the Rocq integration has a strongly comparable proof capability.

In terms of proof ability I believe that Rocq provides a more mature and capable interface than Agda. Taking as example the Wind Controller example above, the proof as written in Agda required a significant portion of additional properties regarding rational numbers (documented in the accompanying `RationalUtils.agda` script, available here). Conversely, the proof in Rocq required no such external lemmas, instead leveraging the vast database of generalised theorems and properties provided by the Mathcomp library. This lead to a more succinct and overall shorter proof script when written in Rocq.

However, I do believe that there are aspects in which the Agda backend presents a more compelling case. Namely, in most examples the compiled Agda scripts match more closely to their source Vehicle specifications. Whereas when compiling to Rocq there can appear some idiosyncrasies where constructs in the source specification do not map one-to-one with constructs in Rocq. These are primarily syntactic differences such as differing notations for indexing and inline return types, this potentially raises the barrier to entry for users of Vehicle to choose Rocq instead of Agda. I believe that this difference is largely down to Vehicle and Agda both being based in the syntax of Haskell, while Rocq is based on Ocaml; thus this may be an unavoidable consequence of the language choice for this project. That being said, potential mitigations for this are documented later in Section 6.3.

As a final note, Agda has some integration with Vehicle's verification cache in an attempt to guarantee that the properies described axiomatically in the specification are only allowed if Vehicle has already verified them. However, the current approach has some major flaws, documented here[4]. In contrast, the Rocq backend currently has no integration with the verification cache; potential approaches for adding this feature are given in Section 6.2.

5.2. Project Management

From a project management standpoint, I believe that this project has been successful. The Gantt chart in Figure 3 outlines the timescales both as predicted in the project progress report and the estimated actual timescales achieved. During the process I encountered unexpected impediments to my progress which took careful planning and due consideration to overcome. These will be discussed further in the following section.

The primary source of blockers during this project orignated from the ongoing development of the `tensor-refactor` branch I had been developing from. At times this meant that my own development had to be halted entirely until a key underlying feature had been fixed. In order to combat this I took the approach of moving later tasks forward in the schedule and making progress on them while the main development stagnated. This was most apparent later in the project schedule and can be seen between weeks 27/01 and 24/03 in Figure 3.

Note also the 'Advanced test case' objective was revised to out-of-scope for this project. This was largely due to the time constraints described above; as well as confidence in the fact that the test suite, along with the useage examples given in Sections 4.4 and 4.5 would be adequate to demonstrate the success of the project. This task would have required significant time familiarising myself with more complex constructs in the Mathcomp library, time which I believe was better spent improving the Rocq backend, the project's primary focus.

The final notable addition to the project Gantt chart is the '*Test suite implementation**' task. This was erroneously excluded from the original Gantt chart, being thought of as part of the larger 'Vehicle to Rocq transpiler' task. In hindsight, this is better represented as its own task as it formed a much larger proportion of time than expected.

In summary, despite some unforseen obstacles to both time management and task prioritisation, this project was successful in keeping reasonably close to the planned schedule.

[4] See https://github.com/vehicle-lang/vehicle/issues/73 for the relevant GitHub issue.

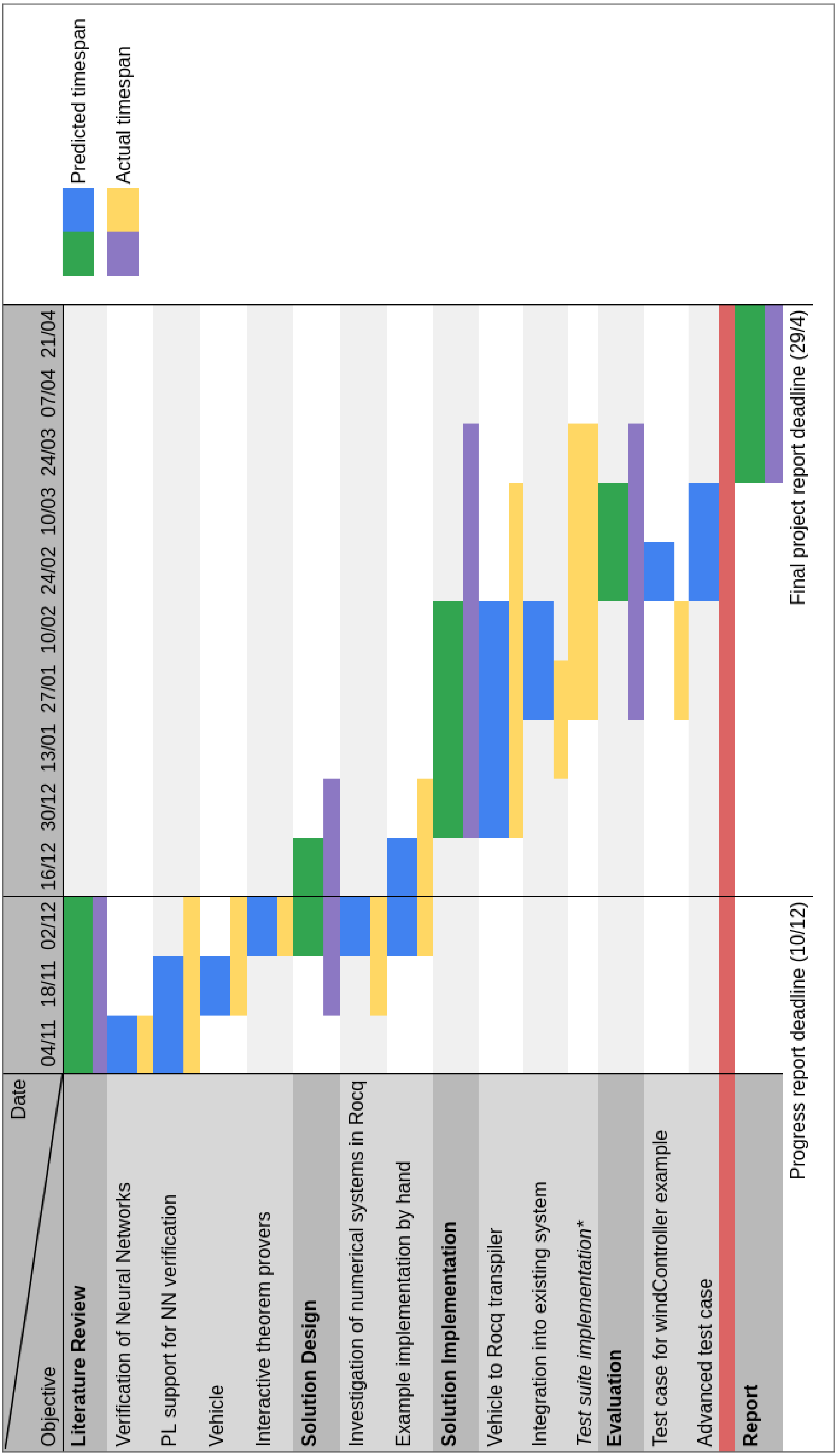

Figure 3: Revised Gantt chart showing predicted and actual project timescales

6. Conclusions and Future Work

In summary, this paper outlines the addition of the interactive theorem prover Rocq as a backend to the Vehicle language. The Rocq backend has been shown to be a capabable feature for verification, being as effective if not more than its Agda counterpart. This addition will widen the problem scope Vehicle is equipted to handle, as well as grow the potential user-base of the tool. The Rocq backend provides a number of positive differences between it and the existing Agda backend, which I hope will allow Vehicle to continue development into a mature and well-featured platform for NN verification.

6.1. Improved Tensor Representation

Section 3.5 discussed the current implementation of Rocq tensors for Vehicle, alongside some existing drawbacks. To recify these issues, future development could focus on integrating a tensor formalisation into Mathcomp. This would provide the following benefits: firstly, it would open up opportunities for users to define properties over the tensor structure directly, manipulating them equally to any other numeric type; this, in turn allows for more idiomatic specifications, as the need for users to universally quantify properties over tensors would be eliminated; finally, migrating the tensor type out of the Vehicle-Rocq library, would reduce its API surface to exclusively standard library definitions, improving maintainability for the project.

6.2. Integration with Verification Cache

In its current form, the Rocq backend has no integration with Vehicle's verification cache (the verification cache can be used to ensure that no previous structures in the verification pipeline have changed, and its existence and validity can be used as a garuntee the correctness of the specified property). It is the responsibility of the user to ensure that the specification is not eroneously violated, thereby invalidating the system proof.

Future work could include integration of this mechanism, possibly via Rocq's annotations feature, to check that the verification cache exists and is valid, aborting compilation otherwise. This would connect the final pieces of the system, giving full confidence to the user that their verification is unimpeachable. However, as part of Vehicle's roadmap, future development may include replacing the verification cache with generated *proof certificates*, making this feature obselete. This follows a recent trend in self-certifying code [33].

6.3. Bridging the Notation Gap

As noted in Section 5.1, one drawback of the Rocq backend is its differing syntax to Vehicle, as a potential barrier to its usage. As a future addition to the Rocq backend, one could leverage its powerful notations system to provide a set of parsing and printing rules, bridging the syntactic gap between the two languages; this would allow users to express properties and proofs in a language they are already familiar with, similiar to how the Agda backend defines the `_!_` infix function for indexing, matching Vehicle's syntax.

Bibliography

[1] M. Daggitt, W. Kokke, N. Ślusarz, R. Atkey, M. Casadio, and E. Komendantskaya, "Vehicle." Accessed: Nov. 07, 2024. [Online]. Available: https://github.com/vehicle-lang/vehicle

[2] B. Neyshabur, S. Bhojanapalli, D. McAllester, and N. Srebro, "Exploring Generalization in Deep Learning." Accessed: Dec. 05, 2024. [Online]. Available: http://arxiv.org/abs/1706.08947

[3] C. Szegedy *et al.*, "Intriguing properties of neural networks." Accessed: Nov. 07, 2024. [Online]. Available: http://arxiv.org/abs/1312.6199

[4] Y. LeCun, C. Cortes, and C. Burges, "MNIST handwritten digit database." Accessed: Nov. 13, 2024. [Online]. Available: https://yann.lecun.com/exdb/mnist/

[5] M. Casadio *et al.*, "Neural Network Robustness as a Verification Property: A Principled Case Study." Accessed: Dec. 09, 2024. [Online]. Available: http://arxiv.org/abs/2104.01396

[6] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, "Safety Verification of Deep Neural Networks." Accessed: Dec. 06, 2024. [Online]. Available: http://arxiv.org/abs/1610.06940

[7] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer, "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks." Accessed: Dec. 06, 2024. [Online]. Available: http://arxiv.org/abs/1702.01135

[8] G. Katz *et al.*, "The Marabou Framework for Verification and Analysis of Deep Neural Networks," in *Computer Aided Verification*, I. Dillig and S. Tasiran, Eds., Cham: Springer International Publishing, 2019, pp. 443–452.

[9] G. Singh, T. Gehr, M. Püschel, and M. Vechev, "An abstract domain for certifying neural networks," *Replication Package for the article: An Abstract Domain for Certifying Neural Networks*, vol. 3, no. POPL, pp. 1–30, Jan. 2019, doi: 10.1145/3290354.

[10] M. N. Müller, G. Makarchuk, G. Singh, M. Püschel, and M. Vechev, "PRIMA: General and Precise Neural Network Certification via Scalable Convex Hull Approximations," *Proceedings of the ACM on Programming Languages*, vol. 6, no. POPL, pp. 1–33, Jan. 2022, doi: 10.1145/3498704.

[11] M. N. Müller, M. Fischer, R. Staab, and M. Vechev, "Abstract Interpretation of Fixpoint Iterators with Applications to Neural Networks," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 786–810, Jun. 2023, doi: 10.1145/3591252.

[12] C. Brix, S. Bak, T. T. Johnson, and H. Wu, "The Fifth International Verification of Neural Networks Competition (VNN-COMP 2024): Summary and Results." Accessed: Apr. 28, 2025. [Online]. Available: http://arxiv.org/abs/2412.19985

[13] C. Brix, M. N. Müller, S. Bak, T. T. Johnson, and C. Liu, "First Three Years of the International Verification of Neural Networks Competition (VNN-COMP)." Accessed: Dec. 06, 2024. [Online]. Available: http://arxiv.org/abs/2301.05815

[14] M. Daggitt *et al.*, "The Vehicle Tutorial: Neural Network Verification with Vehicle," pp. 1–5. doi: 10.29007/5s2x.

[15] W. Kokke, E. Komendantskaya, D. Kienitz, R. Atkey, and D. Aspinall, "Neural Networks, Secure by Construction," in *Programming Languages and Systems*, B. C. d. S. Oliveira, Ed., Cham: Springer International Publishing, 2020, pp. 67–85. doi: 10.1007/978-3-030-64437-6_4.

[16] J. Girard-Satabin, M. Alberti, F. Bobot, Z. Chihani, and A. Lemesle, "CAISAR: A platform for Characterizing Artificial Intelligence Safety and Robustness." Accessed: Nov. 07, 2024. [Online]. Available: http://arxiv.org/abs/2206.03044

[17] T. Freeman and F. Pfenning, "Refinement types for ML," *SIGPLAN Not.*, vol. 26, no. 6, pp. 268–277, May 1991, doi: 10.1145/113446.113468.

[18] W. Kokke, "wenkokke/lazuli." Accessed: Nov. 24, 2024. [Online]. Available: https://github.com/wenkokke/lazuli

[19] A. Lemesle, J. Lehmann, and T. L. Gall, "Neural Network Verification with PyRAT." Accessed: Apr. 28, 2025. [Online]. Available: http://arxiv.org/abs/2410.23903

[20] CEA List, "caisar · GitLab." Accessed: Nov. 24, 2024. [Online]. Available: https://git.frama-c.com/pub/caisar

[21] N. Ślusarz, E. Komendantskaya, M. L. Daggitt, R. Stewart, and K. Stark, "Logic of Differentiable Logics: Towards a Uniform Semantics of DL." Accessed: Nov. 24, 2024. [Online]. Available: http://arxiv.org/abs/2303.10650

[22] M. L. Daggitt, W. Kokke, R. Atkey, L. Arnaboldi, and E. Komendantskya, "Vehicle: Interfacing Neural Network Verifiers with Interactive Theorem Provers." Accessed: Nov. 07, 2024. [Online]. Available: http://arxiv.org/abs/2202.05207

[23] H. Barendregt and H. Geuvers, "Proof-Assistants Using Dependent Type Systems," vol. 2, 2001, pp. 1149–1238. doi: 10.1016/B978-044450813-3/50020-5.

[24] H. Geuvers, "Proof assistants: History, ideas and future," *Sadhana*, vol. 34, no. 1, pp. 3–25, Feb. 2009, doi: 10.1007/s12046-009-0001-5.

[25] Agda Developers, "Agda." Accessed: Apr. 28, 2025. [Online]. Available: https://agda.readthedocs.io/

[26] C. Cohen *et al.*, "math-comp/math-comp: The Mathematical Components Library 2.3.0." [Online]. Available: https://doi.org/10.5281/zenodo.14237175

[27] L. C. Cordeiro *et al.*, "Neural Network Verification is a Programming Language Challenge." [Online]. Available: https://arxiv.org/abs/2501.05867

[28] G. Gonthier, "A computer-checked proof of the Four Color Theorem," Mar. 2023. [Online]. Available: https://inria.hal.science/hal-04034866

[29] T. G. Kolda and B. W. Bader, "Tensor Decompositions and Applications," *SIAM Review*, vol. 51, no. 3, pp. 455–500, 2009, doi: 10.1137/07070111X.

[30] S. Lang, *Algebra*, vol. 211. in Graduate Texts in Mathematics, vol. 211. New York, NY: Springer, 2002. doi: 10.1007/978-1-4613-0041-0.

[31] M. C. Feathers, *Working effectively with legacy code.* in Robert C. Martin series. Upper Saddle River, NJ: Prentice Hall Professional Technical Reference, 2005. [Online]. Available: https://learning.oreilly.com/library/view/~/0131177052/?ar?orpq&email=^u

[32] D. Bank, N. Koenigstein, and R. Giryes, "Autoencoders," *Machine Learning for Data Science Handbook: Data Mining and Knowledge Discovery Handbook.* Springer International Publishing, Cham, pp. 353–374, 2023. doi: 10.1007/978-3-031-24628-9_16.

[33] K. S. Namjoshi and L. D. Zuck, "Program Correctness through Self-Certification," *Commun. ACM*, vol. 68, no. 2, pp. 74–84, Jan. 2025, doi: 10.1145/3689624.

Appendix

A. Original Project Brief

This project concerns the neural network specification language Vehicle. Taken from the repository's `README`: "Vehicle is a system for embedding logical specifications into neural networks. At its heart is the Vehicle specification language, a high-level, functional language for writing mathematically-precise specifications for your networks." The tool can be used to compile specifications to various backends; to loss functions to aid with training models; to network verifiers such as Marabou and to Interactive Theorem Provers (ITPs) to allow formal verification of larger systems. In its current iteration, ITP backend support is limited to Agda.

Goals

The goal of this project is to add support for other ITPs as a backend to the Vehicle language. This will require implementing a compiler between Vechicle specifications and the ITPs theorem language demonstrating the desired properties, as well as writing comprehensive test cases to give confidence in the correctness of the implementation.

Scope

This project's scope is limited to the implementation of the Coq ITP backend. Coq has been chosen as a mature, capable, and developer friendly ITP widely used in similar research.

Motivation

In its current form, compilation to Adga sets some limitations on the expressiveness of specifications, primarily due to its lack of library support for common fields such as Calculus or Analysis. Integration of a Coq backend would remedy this issue as it has a mature ecosystem of libraries for these topics, allowing users to create more natural specifications for cyber-physical systems.

Listing 17: Original project brief (please note that the 'Coq' language has since been renamed to 'Rocq' following the outset of this project.)

B. Vehicle DSL

B.1 Types

Type	Explanation
`Bool`	$\{\mathrm{TRUE}, \mathrm{FALSE}\}$
`Index`	$\{m \mid m \in \mathbb{N} \land m < n\}$ natural numbers between 0 (inclusive) and n (exclusive).
`Nat`	Equivalent to $\mathbb{N}$
`Rat`	Equivalent to $\mathbb{Q}$
`List A`	A sequence of elements of type `A` with unknown length.
`Tensor A [d1, ..., dn]`	A tensor of elements of type `A` with dimensions $d_1 \times \ldots \times d_n$

Table 2: Types available in the Vehicle language [14, 2.2.1].

B.2 Directives

Directive	Explanation	Example

`@network`	This directive is used to declare a neural network which is treated like a black box function.	Haskell
		`@network`
		`network myNetwork`
		`  : Tensor Rat [16, 16]`
		`  -> Tensor Rat [5]`

`@dataset`	This directive is used to introduce a dataset from an external source.	Haskell
		`@dataset`
		`dataset myDataset : Tensor`
		`Rat [100, 4]`

`@parameter`	This directive introduces a value that must be provided at compile time.	Haskell
		`@parameter`
		`parameter myParameter : Rat`

`@property`	This directive declares the property of the network we would like to verify.	Haskell
		`@property`
		`positive : Bool`
		`positive = forall i. network`
		`i > 0`

Table 3: Directives available in the Vehicle language [14].

C. Wind Controller Example

C.1 `WindControllerSpec.v` using Rationals

Rocq

```
(* Generated by Vehicle *)

Require Import Coq.Vectors.Vector.
Import VectorNotations.
Require Import QArith.

Definition InputVector : Type := t Q 2.

Definition currentSensor : Fin.t 2 := Fin.F1.
Definition previousSensor : Fin.t 2 := Fin.FS (Fin.F1).

Definition OutputVector : Type := t Q 1.

Definition velocity : Fin.t 1 := Fin.F1.


Parameter controller : InputVector -> t Q 1.

Definition normalise (x: InputVector) : InputVector :=
    map (fun i => (i + 4) / 8) x.


Definition safeInput (x: InputVector) : Prop :=
    forall i, -3.25 <= nth x i <= 3.25.

Definition safeOutput (x: InputVector) : Prop :=
    -1.25 < (controller x) [@ Fin.F1] + 2 * (nth x currentSensor) - (nth x previousSensor) < 1.25.

Axiom safe : forall x, safeInput x -> safeOutput x.
```

Listing 18: Hand written property specification using rationals, emulating the format that Vehicle should generate. (`WindControllerSpec.v`)

C.2 `SafetyProof.v` using Rationals

Rocq

```
Require WindControllerSpec.
Require Import QArith.
Require Import Coq.QArith.Qabs.
Require Import Coq.Vectors.Vector.
Import VectorNotations.
Require Import Lra.
Require Import Lqa.

(* ----------------------------------------------*)
(* Setup *)

Definition toTensor (x : Q) (y : Q) : t Q 2 := [ x ; y ].

Definition roadWidth : Q := 3.
Definition maxWindShift : Q := 1.
Definition maxSensorError : Q := 1#4.


Lemma roadWidth_ge_0 :
    roadWidth >= 0.
Proof. discriminate. Qed.

Lemma maxWithShift_ge_0 :
    maxWindShift >= 0.
Proof. discriminate. Qed.

Lemma maxSensorError_ge_0 :
    maxSensorError >= 0.
Proof. discriminate. Qed.

(* ----------------------------------------------*)
(* Model Data *)

Record State :=
    { windSpeed : Q
    ; position : Q
    ; velocity : Q
    ; sensor : Q
    }.

Record Observation :=
    { windShift : Q
    ; sensorError : Q
    }.

(* ----------------------------------------------*)
(* Model Transitions *)

Definition initialState : State :=
```

```
    {| windSpeed := 0
    ; position := 0
    ; velocity := 0
    ; sensor := 0
    |}.

Definition controller (x : Q) (y : Q) : Q :=
    (WindControllerSpec.controller (WindControllerSpec.normalise
    (toTensor x y))) [@ Fin.F1].

Definition nextState (o : Observation) (s : State) : State :=
    let newWindSpeed := s.(windSpeed) + o.(windShift) in
    let newPosition := s.(position) + s.(velocity) + newWindSpeed in
    let newSensor := newPosition + o.(sensorError) in
    let newVelocity := s.(velocity) + controller newSensor s.(sensor) in
    {| windSpeed := newWindSpeed
    ; position := newPosition
    ; velocity := newVelocity
    ; sensor := newSensor
    |}.

Definition finalState (xs : list Observation) : State :=
    Coq.Lists.List.fold_right nextState initialState xs.

(* ----------------------------------------------*)
(* Definition of Correctness *)

Definition nextPosition_windShift (s : State) : Q :=
    s.(position) + s.(velocity) + s.(windSpeed).

Definition onRoad (s : State) : Prop :=
    Qabs s.(position) <= roadWidth.

Definition safeDistanceFromEdge (s : State) : Prop :=
    Qabs (nextPosition_windShift s) < roadWidth - maxWindShift.

Definition accurateSensorReading (s : State) : Prop :=
    Qabs (s.(position) - s.(sensor)) <= maxSensorError.

Definition sensorReadingNotOffRoad (s : State) : Prop :=
    Qabs s.(sensor) <= roadWidth + maxSensorError.

Definition safeState (s : State) : Prop :=
    safeDistanceFromEdge s
    /\ accurateSensorReading s
    /\ sensorReadingNotOffRoad s.

Definition validObservation (o : Observation) : Prop :=
    Qabs o.(sensorError) <= maxSensorError
    /\ Qabs o.(windShift) <= maxWindShift.

(* ----------------------------------------------*)
```

```
(* Proof of Correctness *)

Theorem initialState_onRoad : onRoad initialState.
Proof.
    unfold onRoad. simpl. apply roadWidth_ge_0.
Qed.

Theorem initialState_safe : safeState initialState.
Proof.
    unfold safeState. split.
    - unfold safeDistanceFromEdge. reflexivity.
    - split.
      + unfold accurateSensorReading. simpl. apply maxSensorError_ge_0.
      + unfold sensorReadingNotOffRoad. simpl. apply Qlt_le_weak. reflexivity.
Qed.

Lemma controller_lem :
    forall x y,
        WindControllerSpec.safeInput (toTensor x y) ->
        Qabs (controller x y + 2 * x - y) < roadWidth - maxWindShift - 3 * maxSensorError.
Proof.
    intros x y H1.
    unfold roadWidth. unfold maxWindShift. unfold maxSensorError.
    assert (H := WindControllerSpec.safe).
    assert (H2 : 3 - 1 - 3 * (1 # 4) = (5 # 4)). reflexivity. rewrite H2. apply Qabs_Qlt_condition.
```

Listing 19: Incomplete safety proof translated from the Agda file into Rocq using rationals [1, examples/windController/agdaProof/SafetyProof.agda]

C.3 `WindControllerSpec.v` using Reals

Rocq

```
(* WARNING: This file was generated automatically by Vehicle *)
(* and should not be modified manually! *)
(* Metadata: *)
(*  - Rocq version: 0.0.0 *)
(*  - Vehicle version: 0.16.0+dev *)

Require Import mathcomp.ssreflect.ssrbool.
Require Import mathcomp.algebra.ssralg.
Require Import mathcomp.ssreflect.ssrnat.
Require Import mathcomp.ssreflect.order.
Require Import mathcomp.ssreflect.fintype.
Require Import mathcomp.ssreflect.seq.
Require Import mathcomp.ssreflect.tuple.
Require Import mathcomp.algebra.zmodp.
Require Import mathcomp.reals.reals.
Require Import vehicle.tensor.
Require Import vehicle.std.
```

```
Import DefaultTupleProdOrder.
Open Scope ring_scope.
Open Scope tensor_scope.
Open Scope order_scope.

Parameter R : realType.

Definition InputVector : Type := tensor R (2%N :: nil).

Definition currentSensor : ordinal 2%N := 0.

Definition previousSensor : ordinal 2%N := 1.

Definition OutputVector : Type := tensor R (1%N :: nil).

Definition velocity : ordinal 1%N := 0.

Parameter controller : InputVector -> OutputVector.

Definition normalise (x : InputVector) : InputVector := foreach (fun i => tnth x i +%t (4 : R) /%t (8 : R)).

Definition safeInput (x : InputVector) : Prop := forallIndex (fun i => (oppt (13 / 4 : R) <= tnth x i) /\ (tnth x i <= (13 / 4 : R))).

Definition safeOutput (x : InputVector) : Prop := let y := tnth (controller (normalise x)) velocity in (oppt (5 / 4 : R) < ((y +%t ((2 : R) *%t tnth x currentSensor)) -%t tnth x previousSensor)) /\ (((y +%t ((2 : R) *%t tnth x currentSensor)) -%t tnth x previousSensor) < (5 / 4 : R)).

Axiom safe : forall x, safeInput x -> safeOutput x.
```

Listing 20: Property specification using reals, compiled from the specification given in Listing 3. (`WindControllerSpec.v`)

C.4 `SafetyProof.v` using Reals

```Rocq
From mathcomp Require Import all_ssreflect all_algebra reals lra.
From mathcomp.algebra_tactics Require Import ring.
Set Implicit Arguments.
Unset Strict Implicit.
Unset Printing Implicit Defensive.
Import Num.Theory GRing.
Require Import vehicle.tensor.

Open Scope ring_scope.

Require WindControllerSpec.

```

```
Notation R := WindControllerSpec.R.

(* ----------------------------------------------*)
(* Setup *)

Definition toTensor (x : R) (y : R) : tensor R [:: 2] := [tuple x ; y ].

Definition roadWidth : R := 3.
Definition maxWindShift : R := 1.
Definition maxSensorError : R := 1/4.

Lemma roadWidth_ge_0 :
    roadWidth >= 0.
Proof. by []. Qed.

Lemma maxWithShift_ge_0 :
    maxWindShift >= 0.
Proof. by []. Qed.

Lemma maxSensorError_ge_0 :
    maxSensorError >= 0.
Proof. by rewrite mulr_ge0// invr_ge0. Qed.

(* ----------------------------------------------*)
(* Model Data *)

Record State :=
    { windSpeed : R
    ; position : R
    ; velocity : R
    ; sensor : R
    }.

Record Observation :=
    { windShift : R
    ; sensorError : R
    }.

(* ----------------------------------------------*)
(* Model Transitions *)

Definition initialState : State :=
    {| windSpeed := 0
    ; position := 0
    ; velocity := 0
    ; sensor := 0
    |}.

Definition controller (x : R) (y : R) : R :=
    tnth (WindControllerSpec.controller (WindControllerSpec.normalise
    (toTensor x y))) 0.

```

```
Definition nextState (o : Observation) (s : State) : State :=
    let newWindSpeed := s.(windSpeed) + o.(windShift) in
    let newPosition := s.(position) + s.(velocity) + newWindSpeed in
    let newSensor := newPosition + o.(sensorError) in
    let newVelocity := s.(velocity) + controller newSensor s.(sensor) in
    {| windSpeed := newWindSpeed
    ; position := newPosition
    ; velocity := newVelocity
    ; sensor := newSensor
    |}.

Definition finalState (xs : seq Observation) : State :=
    foldr nextState initialState xs.

(* ---------------------------------------------*)
(* Definition of Correctness *)

Definition nextPosition_windShift (s : State) : R :=
    s.(position) + s.(velocity) + s.(windSpeed).

Definition onRoad (s : State) : Prop :=
    `| s.(position) | <= roadWidth.

Definition safeDistanceFromEdge (s : State) : Prop :=
    `| nextPosition_windShift s | < roadWidth - maxWindShift.

Definition accurateSensorReading (s : State) : Prop :=
    `| s.(position) - s.(sensor) | <= maxSensorError.

Definition sensorReadingNotOffRoad (s : State) : Prop :=
    `| s.(sensor) | <= roadWidth + maxSensorError.

Definition safeState (s : State) : Prop :=
    safeDistanceFromEdge s
    /\ accurateSensorReading s
    /\ sensorReadingNotOffRoad s.

Definition validObservation (o : Observation) : Prop :=
    `| o.(sensorError) | <= maxSensorError
    /\ `| o.(windShift) | <= maxWindShift.

(* ---------------------------------------------*)
(* Proof of Correctness *)

Theorem initialState_onRoad : onRoad initialState.
Proof. by rewrite /onRoad normr0. Qed.

Theorem initialState_safe : safeState initialState.
Proof.
    repeat apply conj.
    rewrite /safeDistanceFromEdge /nextPosition_windShift/= !addr0
    normr0 /roadWidth /maxWindShift. by lra.
```

```
    rewrite /accurateSensorReading /nextPosition_windShift/= subr0 normr0 /maxSensorError. by lra.
    rewrite /sensorReadingNotOffRoad normr0 /roadWidth /maxSensorError. by lra.
Qed.

Lemma controller_lem :
    forall x y,
        `| x | <= roadWidth + maxSensorError ->
        `| y | <= roadWidth + maxSensorError ->
        `| controller x y + 2 * x - y | < roadWidth - maxWindShift - 3 * maxSensorError.
Proof.
    move=> x y Hx Hy.
    rewrite /controller.
    replace (roadWidth - maxWindShift - 3 * maxSensorError) with (125 / 100 : R);
        last by rewrite /roadWidth /maxWindShift /maxSensorError; lra.
    rewrite real_lter_norml//=.
    replace (2 * x) with (2 * tnth (toTensor x y) WindControllerSpec.currentSensor);
        last by rewrite /WindControllerSpec.currentSensor.
    replace (- y) with (- tnth (toTensor x y) WindControllerSpec.previousSensor);
        last by rewrite /WindControllerSpec.previousSensor.
    replace 0 with WindControllerSpec.velocity; last by [].
    apply (WindControllerSpec.safe (toTensor x y)).
    rewrite /toTensor /WindControllerSpec.safeInput.
    replace (325 / 100) with (roadWidth + maxSensorError); last by rewrite /roadWidth /maxSensorError; lra.
    elim; case. by move: Hx; rewrite real_lter_norml; last by apply num_real.
    case. move=> i. by move: Hy; rewrite real_lter_norml; last by apply num_real.
    by auto.
Qed.

Lemma valid_imp_nextState_accurateSensor :
    forall o, validObservation o ->
    forall s, accurateSensorReading (nextState o s).
Proof.
    move=> o [H1 H2] s. rewrite /accurateSensorReading/=.
    set (v := position s + velocity s + (windSpeed s + windShift o)).
    rewrite opprD addrA addrC addrN addr0 normrN. by apply: H1.
Qed.


Lemma valid_and_safe_imp_nextState_onRoad :
    forall o, validObservation o ->
    forall s, safeState s ->
    onRoad (nextState o s).
Proof.
```

```
    rewrite /validObservation /safeState. move=> o [Hsensor Hws] s [Hsafedist [Haccsensor Hsenonroad]].
    rewrite /onRoad/= addrA.
    apply /Order.le_trans; first by apply ler_normD.
    rewrite -lerBrDr.
    rewrite /safeDistanceFromEdge in Hsafedist.
    apply /Order.le_trans; first apply Order.POrderTheory.ltW; first by apply: Hsafedist.
    by apply lerB.
Qed.

Lemma valid_and_safe_imp_nextState_sensorReading_not_off_road :
    forall s, safeState s ->
    forall o, validObservation o ->
    sensorReadingNotOffRoad (nextState o s).
Proof.
    move=> s Hs o Ho.
    pose HnextOnRoad := valid_and_safe_imp_nextState_onRoad Ho Hs.
    move: Ho HnextOnRoad. rewrite /onRoad/=. move=> [Hsensor HwindShift] HnextOnRoad.
    rewrite /sensorReadingNotOffRoad/=.
    apply /Order.le_trans; first by apply ler_normD.
    by apply lerD.
Qed.

Lemma valid_and_safe_imp_nextState_safeDistanceFromEdge :
    forall o, validObservation o ->
    forall s, safeState s ->
    safeDistanceFromEdge (nextState o s).
Proof.
    move=> o Ho s Hs.
    pose HnextSensorOnRoad := valid_and_safe_imp_nextState_sensorReading_not_off_road Hs Ho.
    move: Ho HnextSensorOnRoad; rewrite /validObservation/sensorReadingNotOffRoad/=. move=> [Hsensor HwindShift] HnextSensorOnRoad.
    rewrite /safeDistanceFromEdge /nextPosition_windShift/=.
    remember (position s + velocity s + (windSpeed s + windShift o) + sensorError o) as x.
    repeat rewrite addrA.
    replace (position s + velocity s + windSpeed s + windShift o + velocity s +
    controller x (sensor s) + windSpeed s + windShift o) with (
        (controller x (sensor s) + 2 * x - sensor s) + (sensor s - position s - sensorError o - sensorError o)
    ); last by lra.
    apply /Order.POrderTheory.le_lt_trans; first by apply ler_normD.
    apply (@Order.POrderTheory.le_lt_trans _ _ (`|controller x (sensor s) + 2 * x - sensor s| + 3 * maxSensorError)).
    apply lerD; first by [].
    apply /Order.POrderTheory.le_trans; first apply ler_normD.
```

```
    replace (3 * maxSensorError) with (2 * maxSensorError + maxSensorError); last by lra.
    apply lerD; last by rewrite normrN.
    apply /Order.POrderTheory.le_trans; first apply ler_normD.
    replace (2 * maxSensorError) with (maxSensorError + maxSensorError); last by lra.
    apply lerD; last by rewrite normrN.
    rewrite -normrN opprD/= opprK addrC. move: Hs => [_ [HaccSensor _]]. move: HaccSensor; apply.
    rewrite -ltrBrDr. apply controller_lem; first by apply HnextSensorOnRoad.
    move: Hs => [_ [_ HsensorOnRoad]]; move: HsensorOnRoad. by apply.
Qed.

Lemma safe_imp_nextState_safe :
    forall s, safeState s ->
    forall o, validObservation o ->
    safeState (nextState o s).
Proof.
    move=> s Hs o Ho. rewrite /safeState.
    split. by apply valid_and_safe_imp_nextState_safeDistanceFromEdge.
    split. by apply valid_imp_nextState_accurateSensor.
    by apply valid_and_safe_imp_nextState_sensorReading_not_off_road.
Qed.

Lemma finalState_safe :
    forall xs, (forall x, Coq.Lists.List.In x xs -> validObservation x) ->
    safeState (finalState xs).
Proof.
    move=> xs H. induction xs.
    by apply initialState_safe.
    apply safe_imp_nextState_safe. apply IHxs. move=> x HI. apply H.
    right. by apply HI. apply (H a). by left.
Qed.

Lemma finalState_onRoad :
    forall xs, (forall x, Coq.Lists.List.In x xs -> validObservation x) ->
    onRoad(finalState xs).
Proof.
    move=> xs H. induction xs.
    by apply initialState_onRoad.
    apply valid_and_safe_imp_nextState_onRoad. apply (H a). by left.
    apply finalState_safe. move=> x HI. apply H. right. by apply HI.
Qed.
```

Listing 21: Complete safety proof derived from the `windController` example in the Vehicle repository [1, `examples/windController/rocqProof/SafetyProof.v`]

D. GitHub Actions Configuration

YAML

```yaml
name: Test Rocq

on:
  workflow_call:

defaults:
  run:
    shell: sh

jobs:
  test-vehicle-rocq:
    strategy:
      matrix:
        os:
          - name: "Linux"
            type: "ubuntu-latest"
            plat: "manylinux_2_17_x86_64.manylinux2014_x86_64"
          - name: "macOS"
            type: "macos-latest"
            plat: "macosx_10_9_x86_64"
        haskell:
          - ghc:
              version: "9.4.8"
            cabal:
              version: "3.10.2.1"
              project-file: "cabal.project.ghc-9.4.8"
              extra-args: ""
        ocaml:
          - version: "5"
        rocq:
          - version: "9.0.0"

    name: rocq / ${{ matrix.os.name }} - Ocaml ${{ matrix.ocaml.version }} - Rocq ${{ matrix.rocq.version }}
    runs-on: ${{ matrix.os.type }}

    steps:
      - uses: actions/checkout@v4

      - name: Setup Haskell
        uses: ./.github/actions/setup-haskell
        with:
          ghc-version: ${{ matrix.haskell.ghc.version }}
          cabal-version: ${{ matrix.haskell.cabal.version }}
          cabal-project-file: ${{ matrix.haskell.cabal.project-file }}
          cabal-project-freeze-file: ${{ matrix.haskell.cabal.project-file }}.freeze

      - name: Setup Ocaml
```

```yaml
        uses: ocaml/setup-ocaml@v3.2.16
        with:
          ocaml-compiler: ${{ matrix.ocaml.version }}

      - name: Install Rocq
        run: opam install coq.${{ matrix.rocq.version }}

      - name: Install libraries
        run: |
          opam repo add coq-released https://coq.inria.fr/opam/released
          opam install ./vehicle-rocq
          opam install coq-mathcomp-ssreflect coq-mathcomp-algebra coq-mathcomp-reals

      - name: Test Vehicle-Rocq interaction
        run: |
          eval $(opam env)
          cabal test                                 \
            vehicle:test:golden-tests                \
            --test-show-details=always               \
            --test-option=--color=always             \
            --test-option=--num-threads=1            \
            --test-option=--allowlist-externals=coqc

```

Listing 22: GitHub Actions workflow specification in-use with the Vehicle repository [1, `.github/workflows/test-integration-rocq.yml`]

E. Autoencoder Example

E.1 Compiled Rocq Specification

```rocq
(* WARNING: This file was generated automatically by Vehicle *)
(* and should not be modified manually! *)
(* Metadata: *)
(*   - Rocq version: 0.0.0 *)
(*   - Vehicle version: 0.16.0+dev *)

Require Import mathcomp.ssreflect.ssrbool.
Require Import mathcomp.algebra.ssralg.
Require Import mathcomp.ssreflect.ssrnat.
Require Import mathcomp.ssreflect.order.
Require Import mathcomp.ssreflect.seq.
Require Import mathcomp.ssreflect.tuple.
Require Import mathcomp.reals.reals.
Require Import vehicle.tensor.
Require Import vehicle.std.
Import DefaultTupleProdOrder.
Open Scope ring_scope.
Open Scope tensor_scope.
```

```
Open Scope order_scope.

Parameter R : realType.

Parameter encode : tensor R (5%N :: nil) -> tensor R (2%N :: nil).

Parameter decode : tensor R (2%N :: nil) -> tensor R (5%N :: nil).

Definition epsilon : tensor R (5%N :: nil) := foreach (fun i => 1 / 10 :
R).

Axiom identity : forallIndex (fun i => forall x, ((tnth x i -%t tnth
epsilon i) <= tnth (decode (encode x)) i) /\ (tnth (decode (encode x)) i
<= (tnth x i +%t tnth epsilon i))).
```

Listing 23: Compiled Autoencoder Rocq specification

F. Documentation Excerpt

```reStructuredText
Rocq
~~~~

The Rocq backend produces a new specification with the specification's
functions lifted to Rocq's :code:`Prop` type. The network properties are
given as axiomatic assumptions within.

The generated spec is closely linked to the popular mathcomp libraries, this
allows for a more capable and expressive language for wider proofs. See the
car example project for a demonstration of its usage.

Limitations
***********

Postulated resources
####################

Similarly to Agda, networks and datasets are expressed as
opaque :code:`Parameter` declarations within Rocq. Hence it is not possible
to evaluate a network within Rocq.

No integration with verification cache
######################################

Currently Rocq does not integrate with Vehicle's verification cache, meaning
that it is up to the user to garuntee that the compiled specification does
not become out of date with the Vehicle spec.

Poor tensor integration with Mathcomp
#####################################
```

```
Currently, tensors are implemented using nested mathcomp tuple types and does
not directly interface with mathcomp's structure hierarchy. This can lead to
issues when considering properties with tensor arithmetic. Users are
encouraged to, when possible, express tensor properties using universal
quantification over indicies. This generally leads to neater proofs.
```

Listing 24: Excerpt from changes to Vehicle's documentation, specifically the source of the Vehicle documentation site at https://vehicle-lang.readthedocs.io/en/stable/

G. Git Diff of Contribution

```
 .github/workflows/test-integration-rocq.yml       |  70 +++
 .gitignore                                        |  12 +
 CONTRIBUTING.md                                   |  17 +-
 README.md                                         |   1 +
 docs/exporting.rst                                |  42 +-
 examples/autoencoderError/README.md               |  22 +
 examples/autoencoderError/rocqProof/Proof.v       |  28 +
 examples/autoencoderError/rocqProof/_CoqProject   |   2 +
 examples/autoencoderError/spec.vcl                |  12 +
 examples/windController/README.md                 |   8 +-
 examples/windController/rocqProof/SafetyProof.v   | 238 ++++++++
 examples/windController/rocqProof/_CoqProject     |   2 +
 vehicle-rocq/Makefile                             |  25 +
 vehicle-rocq/_CoqProject                          |   5 +
 vehicle-rocq/std.v                                |  28 +
 vehicle-rocq/tensor.v                             |  84 +++
 vehicle-rocq/vehicle-rocq.opam                    |  26 +
 vehicle/src/Vehicle/Backend/Agda/Interact.hs      |  23 +-
 vehicle/src/Vehicle/Backend/Prelude.hs            |  28 +-
 vehicle/src/Vehicle/Backend/Rocq.hs               |   7 +
 vehicle/src/Vehicle/Backend/Rocq/Compile.hs       | 614 ++++++++++++++++++
 vehicle/src/Vehicle/Backend/Rocq/Interact.hs      |  25 +
 vehicle/src/Vehicle/Compile.hs                    |   5 +
 vehicle/src/Vehicle/Prelude/IO.hs                 |   7 +-
 vehicle/src/Vehicle/Verify/QueryFormat/Marabou.hs |   2 +-
 vehicle/src/Vehicle/Verify/QueryFormat/VNNLib.hs  |   2 +-
 vehicle/tests/golden/compile/acasXu/Rocq.v.golden | 110 ++++
 vehicle/tests/golden/compile/acasXu/test.json     |  15 +
 vehicle/tests/golden/compile/andGate/Rocq.v.golden |  33 ++
 vehicle/tests/golden/compile/andGate/test.json    |  14 +
 .../golden/compile/autoencoderError/Rocq.v.golden |  27 +
 .../golden/compile/autoencoderError/test.json     |  14 +
 .../golden/compile/dogsHierarchy/Rocq.v.golden    |  57 ++
 .../tests/golden/compile/dogsHierarchy/test.json  |  14 +
 .../tests/golden/compile/help/compile.out.golden  |   2 +-
 .../tests/golden/compile/help/export.out.golden   |   3 +-
 .../tests/golden/compile/increasing/Rocq.v.golden |  25 +
 vehicle/tests/golden/compile/increasing/test.json |  14 +
 .../golden/compile/mnist-robustness/test.json     |  14 +
```

```
 .../golden/compile/monotonicity/Rocq.v.golden      |  25 +
 .../tests/golden/compile/monotonicity/test.json    |  14 +
 .../golden/compile/reachability/Rocq.v.golden      |  21 +
 .../tests/golden/compile/reachability/test.json    |  14 +
 .../golden/compile/simple-arithmetic/Rocq.v.golden |  30 +
 .../golden/compile/simple-arithmetic/test.json     |  14 +
 .../compile/simple-constantInput/Rocq.v.golden     |  25 +
 .../golden/compile/simple-constantInput/test.json  |  14 +
 .../tests/golden/compile/simple-if/Rocq.v.golden   |  30 +
 vehicle/tests/golden/compile/simple-if/test.json   |  14 +
 .../golden/compile/simple-index/Rocq.v.golden      |  29 +
 .../tests/golden/compile/simple-index/test.json    |  14 +
 .../tests/golden/compile/simple-let/Rocq.v.golden  |  42 ++
 vehicle/tests/golden/compile/simple-let/test.json  |  14 +
 .../golden/compile/simple-pruneDecls/Rocq.v.golden |  25 +
 .../golden/compile/simple-pruneDecls/test.json     |  14 +
 .../golden/compile/simple-quantifier/Rocq.v.golden |  31 ++
 .../golden/compile/simple-quantifier/test.json     |  14 +
 .../compile/simple-quantifierIn/Rocq.v.golden      |  30 +
 .../golden/compile/simple-quantifierIn/test.json   |  14 +
 .../golden/compile/simple-tensor/Rocq.v.golden     |  39 ++
 .../tests/golden/compile/simple-tensor/test.json   |  14 +
 .../tests/golden/compile/simple-vector/test.json   |  14 +
 .../golden/compile/windController/Rocq.v.golden    |  43 ++
 .../tests/golden/compile/windController/test.json  |  14 +
 vehicle/vehicle.cabal                              |   3 +
 65 files changed, 2198 insertions(+), 34 deletions(-)
```

Listing 25: Git Diff between the `rocq-backend` and `tensor-refactor` branches (see https://github.com/vehicle-lang/vehicle/compare/tensor-refactor...rocq-backend for full diff contents)

H. Project Archive Contents

```
archive
├── diff.txt // full diff of changes at time of submission
└── vehicle // commit: 1283b162a7695a43725e631391befb6e618602b4
```