Vehicle: A High-Level Language for Embedding Logical Specifications in Neural Networks

Matthew L. Daggitt[1], Wen Kokke[2], Robert Atkey[2], Luca Arnaboldi[3], and Ekaterina Komendantskaya[1]

[1] Department of Computer Science, Heriot-Watt University, Edinburgh, UK
{m.daggit,e.komendantskaya}@hw.ac.uk
[2] Department of Computer and Information Sciences, University of Strathclyde, Glasgow, UK
{robert.atkey,wen.kokke}@strath.ac.uk
[3] School of Informatics, University of Edinburgh, Edinburgh, UK
luca.arnaboldi@ed.ac.uk

Abstract. Verification of neural network specifications is currently an active field of research in automated theorem proving. However, the actual act of verification is merely one step in the process of constructing a verified network. Prior to verification the specification should influence the training of the network, and afterwards users may want to export the verified specification to other verification environments in order to prove a specification about a larger system that uses the network. Currently there is little consensus on how best to connect these different stages.
In this talk we will describe our proposed solution to this problem: the Vehicle specification system. Vehicle allows the user to write a single specification in a high-level human readable form, and the Vehicle compiler then compiles it down to different targets, including training frameworks, verifiers and interactive theorem provers. In this talk we will discuss the various design decisions involved in the specification language itself and hope to solicit feedback from the verification community.

Keywords: Neural networks · Verification · Agda · Marabou

1 Introduction

Significant progress has been made in the last decade on embedding specifications into neural networks, including in areas such as training [7, 8], counter-example search [3, 5] and verification [11, 15]. However, none of the tools in these areas are designed to interact with each other, and users are forced to repeatedly rewrite the same specification in a series of disparate formats. As well as being inconvenient, this increases the likelihood of logical inconsistencies between the different representations of the same specification. Additionally, there is no support for integrating formally verified specifications into interactive theorem provers. We believe that such integration is an important step to being able to formally prove the correctness of larger software developments that contain neural networks sub-components.

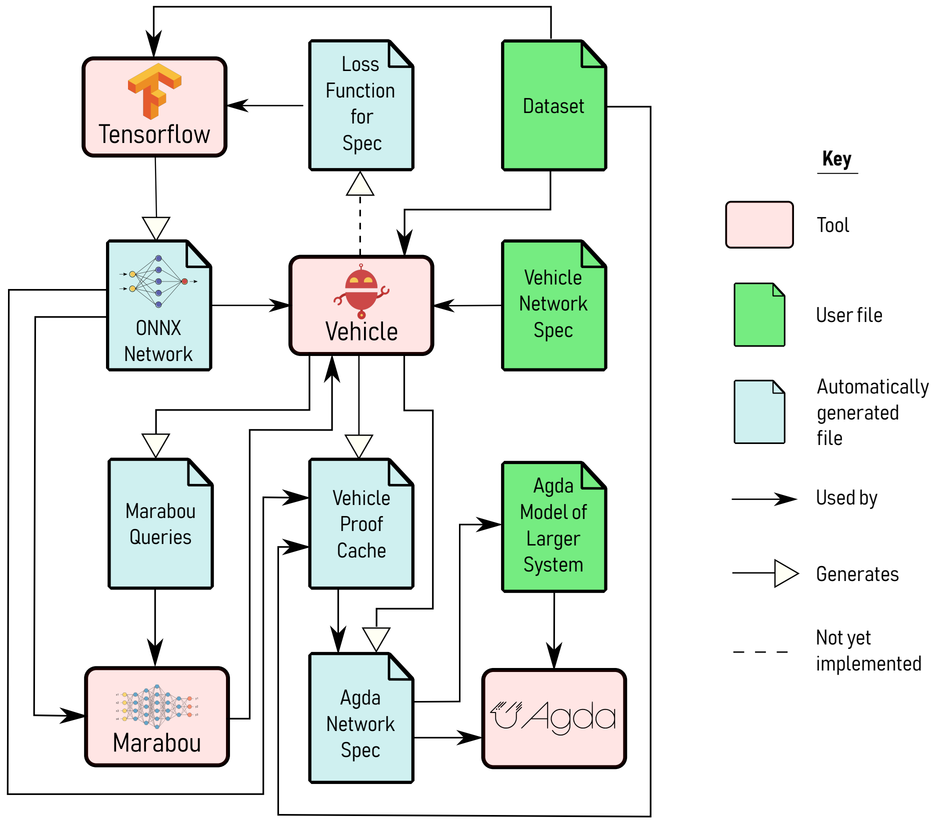

Fig. 1: The architecture of a proof about a neural network enhanced system.

Our proposed system, Vehicle, aims to solve these problems by having the user write only a single definitive high-level specification. This specification is then automatically compiled down to the various tools as required. The full proposed architecture of an example Vehicle project is shown in Figure 1.

The current implementation of the Vehicle compiler targets the neural network verifier Marabou [11] and the interactive theorem prover Agda [13]. Compilation of specifications to loss functions for training is still a work in progress. The system has been designed to be modular, so adding support for new training, verification and theorem proving tools should be relatively straightforward.

While the implementation of the Vehicle compiler contains several novel programming language challenges, in this talk we will focus on the design of the specification language itself. It is important to emphasise this language is supposed to be a high-level and human-readable. This contrasts it to other similar works such as VNNLib [2] and DNNV [14], whose primary focus is to provide a unified input format for verifiers.

We identify the following design principles that we believe that the existing input formats for training and verification fail to satisfy:

P1 Specifications should be comprehensible by experts in the problem domain who have little knowledge of programming, rather than solely by verification experts.
P2 The size of a specification should scale with its inherent complexity, rather than the number of input and output dimensions of the network.
P3 The specification language should be independent of any particular network format, training framework, verifier etc.
P4 Every specification should have a rigorous mathematical semantics associated with it. Not only does this allow the compilation procedure to be formally proved correct, it is also a vital pre-requisite in order to be able to export the specifications to interactive theorem provers.
P5 The language should be expressive enough to cover all possible specifications of interest, even those that aren't supported by current verifiers. For example, specifications that involve multiple networks, or multiple applications of the same network.

This extended abstract is structured as follows. In Section 2 we discuss the design of the language itself. In Section 3, we use it to demonstrate how the famous robustness specification for the MNIST dataset may be encoded. Finally, in Section 4, we discuss our planned future work. We also include two additional example specifications in the appendices: ACAS Xu [10] and a simple autonomous car controller. The tool itself, along with documentation and further examples is available online [6].

2 The specification language

Principle P4 in Section 1 stated that it is important that every well-typed specification has a clear mathematical semantics. Consequently, the language uses a Haskell-style functional paradigm with a statically-checked, dependent type system. The dependent types allow the user to specify the size of tensors in the type-system itself, e.g. the type of 10×10 matrices of rational numbers. This in turn allows type-safe indexing into vectors and tensors. Consequently every well-typed Vehicle program is guaranteed to be free of run-time errors that would interfere with assigning every specification a semantics. We decided against creating a deeply embedded DSL within Haskell itself for two reasons. Firstly we relinquish significant control over the syntax, and secondly Haskell's type-system is not powerful enough to encode the dependent types discussed above.

We will now describe some of the main features of this language. The full BNF grammar and further documentation for the language can be found online [6].

2.1 Basic types

The Vehicle language contains the following basic types:

- `Bool` - booleans
  - Operations: `and`, `or`, `=>`, `not`, `if ... then ... else ...`, `==`, `!=`
- `Index n` - natural numbers between 0 (inclusive) and n (exclusive).
  - Used for safe indexing into tensors. For example, only the values 0 and 1 have type `Index 2`.
  - Operations: `==`, `!=`, `<=`, `>=`, `<`, `>`
- `Nat` - natural numbers
  - Operations: `==`, `!=`, `<=`, `>=`, `<`, `>`, `+`, `*`
- `Int` - integer numbers
  - Operations: `==`, `!=`, `<=`, `>=`, `<`, `>`, `+`, `*`, `-`
- `Rat` - rational numbers
  - Operations: `==`, `!=`, `<=`, `>=`, `<`, `>`, `+`, `*`, `-`, `/`

We note that recent work has shown that the mismatch between rational/real number and floating point semantics leads to soundness problems in verification [9]. As discussed further in Section 4, in the medium-term we aim to replace `Rat` with a set of variable-precision floating point types.

Next there are two container types:

- `List A` - a list of elements of type `A`
  - Used for sequences of data for which one either doesn't care about or don't know the length of.
  - Operations: `==`, `!=`, `map`, `fold`
- `Tensor A [d1, ..., dn]` - a tensor of elements of type A with dimensions $d1 \times ... \times d_n$.
  - Used for data for which it is important to know the size of. Due to the dependently typed-nature of the language, the dimensions can themselves be arbitrary expressions.
  - Operations: `==`, `!=`, `map`, `fold`, `!`

The lookup operation `!` for tensors deserves a special mention. If its first argument is a tensor whose first dimension is `n`, then the index argument must have type `Index n`. This allows the type-system to eliminate the possibility of runtime out-of-bounds exceptions. For example the expression `[0.1, 0.2, 0.3] ! 2` is well-typed, with `1` being given the type `Index 3`, and evaluates to `0.3`. In contrast, the expression `[0.1, 0.2, 0.3] ! 3` is not well-typed as `3` is out of bounds and therefore not a valid element of the type `Index 3`.

2.2 Functions

Functions make up the backbone of the language. Function application is written by juxtaposition, i.e. `f x` represents the function `f` applied to input `x`. The function type is written `A -> B` where `A` is the input type and `B` is the output type. For example `Tensor Rat [10,10] -> Bool` is the type of functions from 10×10 matrix of rational numbers to a single boolean value. However, as the language is dependently typed, the type `B` may depend on the value passed for

A. For example (n : Nat) -> Tensor Rat [n] is a function that takes in a natural number n and returns a vector of rationals of length n.

New functions can be introduced in two ways. Firstly, anonymous lambda functions can be declared using the \x -> notation, e.g. \x -> x + 2 is a function that adds 2 to its input. Alternatively, the same function could be given a name and an explicit type as follows:

```
add2 : Nat -> Nat
add2 x = x + 2
```

and can then be used later in the specification by referring to this name. As will be demonstrated in Section 3, the ability to name functions allows subcomponents of the specification to be given names meaningful within the specific problem domain. This is an example of principle P1 in action.

2.3 Network Declarations

The language models neural networks as black box functions between tensors, and they can be declared as follows using the network keyword:

```
network myNetwork : Tensor Rat [28, 28] -> Tensor Rat [10]
```

The network can then be used as any other function would be, by applying myNetwork to arguments, e.g.

```
lastOutputPositive : Tensor Rat [28, 28] -> Bool
lastOutputPositive x = myNetwork x ! 9 > 0
```

At the moment Vehicle only supports networks with a single input and output node, i.e. type of a network must be of the form Tensor A [m] -> Tensor B [n] where A and B are numeric types and m and n are known constants. We hope to relax this constraint in the near future.

The actual implementation of the network is provided at compilation time, in the form of a path to the underlying file, using the `--network` command line option. Currently Vehicle supports the ONNX network format [1]. This mechanism for declaring and providing networks fulfils principle P3, as the specification is fully decoupled from the implementation of the network.

2.4 Dataset Declarations

Sometimes a specification may want to reference an external source of data. These may be introduced using the using the dataset keyword:

```
dataset myDataset : Tensor Rat [10, 784]
```

Datasets are restricted to being a container type (i.e. List or Tensor). Once declared, datasets can be used as any other named List or Tensor would be, e.g.

```
robust : Bool
robust = forall x in myDataset . robustAround x
```

Again, the dataset is only provided by the user at compile time using the *--dataset* command line option. At the moment Vehicle supports the IDX [12] format.

2.5 Parameter Declarations

Sometimes the user may not want to hard-code a specific value in the specification, but instead provide it at compile time. For example, the value may not be known ahead of time, or you might want to reuse the same spec with multiple different values e.g. assign different values for epsilon in a robustness specification. This can be achieved using the `parameter` keyword:

```
parameter myParameter : Rat
```

Like networks and datasets, parameters are passed in at compile time via the *--parameter* command line option.

2.6 Quantifiers

Vehicle supports both universal and existential quantifiers using the `forall` and `exists` keywords respectively which bring a new named variable into scope to the right of the dot, e.g.

```
property1 : Bool
property1 = forall x . lastOutputPositive x

property2 : Bool
property2 = exists x . forall i . myNetwork x ! i > 0.5
```

Although they can be manually annotated, the compiler can automatically infer the types of the quantified variables, `Tensor Rat [28, 28]` for `x` and `Index 10` for `i`, from their use.

As well as quantifying over variables of an arbitrary type as above, one can also quantify over the elements contained within a tensor or a list, using the `forall ... in` syntax e.g.

```
normalised : Tensor Rat [10] -> Bool
normalised tensor = forall x in tensor . 0 <= x <= 1
```

Note that these quantifiers fulfil principle P2, in that the specification is independent of the size of the input tensor `x`. Changing the dimensions of the input tensor for `myNetwork` from `[28, 28]` to `[280, 280]` would not change definition of `property2` at all. In contrast, if `property2` were to be written in the Marabou input query format, this change would increase the size of the specification by a factor of 100, as the constraints have to be specified individually over each input variable.

2.7 Type synonyms

Although the builtin types are sufficient to write a wide range of properties, specifications can often be made more readable by using the `type` keyword to define domain-specific synonyms for types that are used repeatedly throughout a specification.

For example, when defining a robustness specification for the MNIST dataset which contains 28x28 greyscale images, in order to avoid having to repeatedly write `Tensor Rat [28, 28]`, one could declare `Image` as a synonym for it and use it as follows:

```
type Image = Tensor Rat [28, 28]

network classify : Image -> Tensor Rat [10]
```

Type synonyms can also have arguments. For example one can declare a new type `Vector` as a synonym for 1-dimensional tensors as follows:

```
type Vector A n = Tensor A [n]
```

2.8 Expressiveness and solver-compatibility

It is hopefully obvious that the expressive power of the Vehicle language is strictly greater than the expressiveness of existing low-level query formats for verifiers, such as that of Marabou [11] or VNNLib [2], or that of training frameworks such as DL2 [7]. For example, using Vehicle it is possible to write specifications involving multiple networks (e.g. for encoding constraints between teacher-student networks), something that isn't commonly supported by the current generation of verifiers. We therefore argue that Vehicle satisfies principle P5.

However, this also raises the question what happens when Vehicle's expressive power is used to write queries that aren't soluble by the solver being targeted. For example, given some neural network `f`, the query:

```
squaredSurjective : Bool
squaredSurjective = forall y . exists x . f (x * x) == y
```

is not soluble by Marabou for two reasons. Firstly, the mixing of the `forall` and `exists` quantifiers cannot be translated into a simple satisfaction problem, and secondly, the specification is non-linear in the variable `x`.

To address this, the compiler has a second hidden type-system that statically tracks both the pattern of quantifiers used and whether the variables are all used strictly linearly. The user never needs to provide manual annotations for this type system, and it works even in the presence of higher-order functions (this level of automation is only possible because recursion is not present in the language). Upon detecting an incompatibility with the solver, the type system provides meaningful error messages to the user pinpointing the exact sequence of definitions that leads to the problem.

3 Case study: epsilon-ball robustness

We will walk through an annotated example of the famous epsilon-ball robustness specification for the MNIST dataset. Further example specifications are available in Appendices A & B.

At a high-level, a neural network is robust if any small perturbation to the network's input results in only a small change to the network's output. We begin by defining a type synonym `Image` for the inputs of the network which are 28×28 pixels.

```
type Image = Tensor Rat [28, 28]
```

However, not every instance of `Image` is a valid input as we will assume that pixel values range can only be in the range between 0 and 1. To encode this, we define a predicate `valid` that takes an image as an argument and requires that all its pixel values are in that range:

```
valid : Image -> Bool
valid x = forall i j . 0 <= x ! i ! j <= 1
```

Note the use of `forall` syntax to quantify over all indices `i` and `j`. The compiler will notice that as they are being used an index into `x`, an `Image`, and therefore infer that they must both be of type `Index 28`. The compiler will therefore automatically expand out this quantification to a conjunction of 784 terms of the form `(0 <= x ! 0 ! 0 <= 1) and ... and (0 <= x ! 27 ! 27 <= 1)`.

Next we define type synonyms for the output. The MNIST dataset has 10 classes: the digits 0 - 9, and the network produces a distribution over these classes. These can be represented using the `Index 10` and the `Tensor Rat [10]` types respectively.

```
type Label = Index 10
type LabelDistribution = Tensor Rat [10]
```

Next we declare the network itself. We call it `mnist` and model it as a function which takes an `Image` and produces a `LabelDistribution`. Each index of the output tensor represents the score assigned to that label.

```
network mnist : Image -> LabelDistribution
```

We then define another predicate `advises` that states that the network advises that image `x` should be classified as label `i` if `i`'s score is higher than the score of any other label:

```
advises : Image -> Label -> Bool
advises x i = forall j . j != i => mnist x ! i > mnist x ! j
```

Note how the network `mnist` is treated as just another function, and is applied to input `x`. Again, the compiler will infer that `j` has type `Index 10` (or equivalently `Label`) and therefore the `forall` will again be expanded to a conjunction by the compiler.

We now turn our attention to defining what it means for the mnist network to be robust. We will use the standard epsilon ball robustness formulation here, where we measure distance using the L_∞ norm on pixel values for the standard MNIST dataset.

The first step is to decide what value to set for our robustness tolerance epsilon which will represent the maximum amount each individual pixel can vary by without changing the classification decision of the network. While we could hard-code a specific value (e.g. 0.1), it is more flexible to declare it as a parameter, which allows its value to be specified by the user at compile time:

```
parameter epsilon : Rat
```

We can now use it to define what it means for an image x to be inside the ball of radius epsilon around some central image, namely that no two pixels differ by more than epsilon:

```
lInfBall : Image -> Image -> Bool
lInfBall centre x = forall i j .
  -epsilon <= (x - centre) ! i ! j <= epsilon
```

Using this definition, we define what it means for the network to be robust around an image x that we know should be classified as label y. Namely, that for any valid input image z that lies within the ball of radius epsilon around x the network should still advise label y when asked to classify z.

```
robustAround : Image -> Label -> Bool
robustAround x y = forall (z : Image) .
  valid z and lInfBall x z => advises z y
```

Note that the quantification over z is fundamentally different from the previous quantifiers that we've used until now. This is because as z has type Image/Tensor Rat [784] which is an infinite set. Such a quantifier cannot be normalised out to a finite set of conjunctions, and this is the part of the specification that needs to be sent to Marabou when verifying it.

However the specification is still not complete, as we haven't declared *which* images the network should be robust around. In theory we only care about the network being robust on the set of images it will encounter during operation. Unfortunately we can't characterise this set of "reasonable" input images. Instead, we take the standard approach of approximating it using the training dataset, and simply ask that the network is robust around images in the training dataset.

To achieve this we first specify parameter n the size of the training dataset. Unlike the earlier parameter epsilon, n is marked as implicit which means that it does not need to be provided manually by the user at compile time but instead the compiler will automatically infer its value. In this case it will be inferred from the training dataset passed.

```
implicit parameter n : Nat
```

We next declare two datasets, the training images and the corresponding training labels.

```
dataset trainingImages : Tensor Image [n]
dataset trainingLabels : Tensor Label [n]
```

Note that we use n to enforce that they are the same size. If the actual datasets passed are not the same size, then the compiler will throw an error when compiling the specification.

We then say that the network is robust if it is robust around every pair of input images and output labels.

```
robust : Tensor Bool [n]
robust = foreach i .
  robustAround (trainingImages ! i) (trainingLabels ! i)
```

Note the use of the `foreach` keyword when quantifying over the index `i` in the dataset. Unlike `forall` which produces a single boolean, `foreach` generates a tensor of booleans. This ensures that Vehicle will report on the verification status of each image in the dataset separately. If `forall` was used, then Vehicle would only report if the network was robust around *every* image in the dataset, a state of affairs which is unlikely to be true.

4 Conclusions

We hope that we have convinced the reader that the language presented above is both expressive enough to capture a wide range of specifications, and is a significant step-up in terms of usability and readability.

Our short-term goals for further development of the system are as follows:

1. Compilation of specifications to loss-functions so that they can be used both in training [7] and efficient counter-example search [5].
2. Augmenting the `Rat` type with a comprehensive set of IEEE-compliant floating point types, which can be used to specify both the format and the precision used on the deployed hardware.
3. A formal denotational semantics for both the high-level specification language and the Marabou query language with which the compilation procedure can be proved correct.

In the longer term, we hope that Vehicle's ability to export verified specifications to interactive theorem provers (something not discussed here) will be used to formally verify the overall safety of large neural-network enhanced software developments.

Acknowledgements This work was funded by the AISEC grant under EPSRC numbers EP/T026952/1, EP/T026960/1, and EP/T027037/1. We would like to thank the Marabou development team for their support and advice with integrating our tool with Marabou.

References

1. Open Neural Network Exchange format, https://onnx.ai/, accessed on 30.01.2022
2. VNNLib format, https://vnnlib.org/, accessed on 01.12.2021
3. Athalye, A., Engstrom, L., Ilyas, A., Kwok, K.: Synthesizing robust adversarial examples. In: International conference on machine learning. pp. 284–293. PMLR (2018)
4. Boyer, R.S., Green, M.W., Moore, J.S.: The use of a formal simulator to verify a simple real time control program. In: Beauty Is Our Business, pp. 54–66. Springer (1990)
5. Chiang, P.Y., Geiping, J., Goldblum, M., Goldstein, T., Ni, R., Reich, S., Shafahi, A.: Witchcraft: Efficient PGD attacks with random step size. In: ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). pp. 3747–3751. IEEE (2020)
6. Daggitt, M.L., Kokke, W.: Vehicle, https://github.com/vehicle-lang/vehicle, accessed on 09.02.2022
7. Fischer, M., Balunovic, M., Drachsler-Cohen, D., Gehr, T., Zhang, C., Vechev, M.: DL2: training and querying neural networks with logic. In: International Conference on Machine Learning. pp. 1931–1941. PMLR (2019)
8. Gou, J., Yu, B., Maybank, S.J., Tao, D.: Knowledge distillation: a survey. International Journal of Computer Vision **129**(6), 1789–1819 (2021)
9. Jia, K., Rinard, M.: Exploiting verified neural networks via floating point numerical error. In: International Static Analysis Symposium. pp. 191–205. Springer (2021)
10. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient smt solver for verifying deep neural networks. In: International Conference on Computer Aided Verification. pp. 97–117. Springer (2017)
11. Katz, G., Huang, D.A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljić, A., et al.: The Marabou framework for verification and analysis of deep neural networks. In: International Conference on Computer Aided Verification. pp. 443–452. Springer (2019)
12. yann lecun, c.c., chris burges: The mnist database, http:yann.lecun.comexdbmnist
13. Norell, U.: Dependently typed programming in Agda. In: International school on advanced functional programming. pp. 230–266. Springer (2008)
14. Shriver, D., Elbaum, S., Dwyer, M.B.: DNNV: A framework for deep neural network verification. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification. pp. 137–150. Springer International Publishing, Cham (2021)
15. Singh, G., Gehr, T., Püschel, M., Vechev, M.: An abstract domain for certifying neural networks. Proceedings of the ACM on Programming Languages **3**(POPL), 1–30 (2019)

A ACAS XU - Safety Conditions

ACAS Xu is a collision avoidance system for autonomous unmanned aircraft, and its verification was first described in the seminal Reluplex paper [10]. In this appendix we demonstrate how the entire specification, consisting of all 10 properties, can be written in a single Vehicle specification file. Unlike the equivalent 43 low-level Marabou queries, the specification is written at a high-level and is understandable by a non-expert. However, these queries are automatically generated from the specification and verified using Vehicle. We take the full specification of the ACAS XU networks from Appendix VI of the paper [10], and the textual description of the properties is taken directly from there.

Initial Setup - we first define the types of the inputs & output of the network and add meaningful names for the indices.

```
type InputVector = Tensor Rat [5]

distanceToIntruder = 0
angleToIntruder    = 1
intruderHeading    = 2
speed              = 3
intruderSpeed      = 4

type OutputVector = Tensor Rat [5]

clearOfConflict = 0
weakLeft        = 1
weakRight       = 2
strongLeft      = 3
strongRight     = 4
```

Then we define the network as a function from inputs to outputs.

```
network acasXu : InputVector -> OutputVector
```

For calculating steering angles we make use of the value of the constant `pi`.

```
pi : Rat
pi = 3.141592
```

We also define a helper function `advises` that states the network chooses output `i` when given the input `x`. We must necessarily provide a finite index that is less than 5 (i.e. of type `Index 5`). The `a!b` operator lookups index `b` in tensor `a`.

```
advises : Index 5 -> InputVector -> Bool
advises i x = forall j . i != j => acasXu x ! i < acasXu x ! j
```

Property 1 - first checks the distance to the intruder and its speed using intruderDistantAndSlower, and assesses whether the intruder is distant and is significantly slower than your own ship, such that the score of a Clear of Conflict (COC) advisory will always be below a certain fixed threshold.

```
intruderDistantAndSlower : InputVector -> Bool
intruderDistantAndSlower x =
  x ! distanceToIntruder >= 55947.691 and
  x ! speed              >= 1145      and
  x ! intruderSpeed      <= 60

property1 : Bool
property1 = forall x . intruderDistantAndSlower x =>
  acasXu x ! clearOfConflict <= 1500
```

Property 2 - states that if the intruder is distant and is significantly slower than the ownship, the score of a COC advisory will never be maximal.

```
property2 : Bool
property2 = forall x . intruderDistantAndSlower x =>
  (exists j . (acasXu x ! j) > (acasXu x ! clearOfConflict))
```

Property 3 - states that if the intruder is directly ahead (directlyAhead) and is moving towards the ownship (movingTowards), the score for COC will not be minimal.

```
directlyAhead : InputVector -> Bool
directlyAhead x =
  1500  <= x ! distanceToIntruder <= 1800 and
  -0.06 <= x ! angleToIntruder    <= 0.06

movingTowards : InputVector -> Bool
movingTowards x =
  x ! intruderHeading >= 3.10  and
  x ! speed           >= 980   and
  x ! intruderSpeed   >= 960

property3 : Bool
property3 = forall x . directlyAhead x and movingTowards x =>
  not (advises clearOfConflict x)
```

Property 4 - states that if the intruder is directly ahead and moving away from the ownship but at a lower speed than that of the ownship, the score for COC will not be minimal.

```
movingAway : InputVector -> Bool
movingAway x =
          x ! intruderHeading == 0   and
  1000 <= x ! speed                  and
  700  <= x ! intruderSpeed   <= 800

property4 : Bool
property4 = forall x . directlyAhead x and movingAway x =>
  not (advises clearOfConflict x)
```

Property 5 - states that if the intruder is near and approaching from the left, that the network advises “strong right”.

```
nearAndApproachingFromLeft : InputVector -> Bool
nearAndApproachingFromLeft x =
  250 <= x ! distanceToIntruder <= 400         and
  0.2 <= x ! angleToIntruder    <= 0.4         and
  -pi <= x ! intruderHeading    <= -pi + 0.005 and
  100 <= x ! speed              <= 400         and
  0   <= x ! intruderSpeed      <= 400

property5 : Bool
property5 = forall x . nearAndApproachingFromLeft x
                       => advises strongRight x
```

Property 6 - states that if the intruder is sufficiently far away, the network advises COC.

```
intruderFarAway : InputVector -> Bool
intruderFarAway x =
  12000   <= x ! distanceToIntruder <= 62000        and
  (- pi   <= x ! angleToIntruder <= -0.7 or 0.7 <=
             x ! angleToIntruder <= pi) and
    -pi   <= x ! intruderHeading    <= -pi + 0.005 and
    100   <= x ! speed              <= 1200         and
    0     <= x ! intruderSpeed      <= 1200

property6 : Bool
property6 = forall x . intruderFarAway x
                       => advises clearOfConflict x
```

Property 7 - states that if the vertical separation is large, the network will never advise a strong turn.

```
largeVerticalSeparation : InputVector -> Bool
largeVerticalSeparation x =
  0    <= x ! distanceToIntruder <= 60760  and
  -pi  <= x ! angleToIntruder    <= pi     and
  -pi  <= x ! intruderHeading    <= pi     and
  100  <= x ! speed              <= 1200   and
  0    <= x ! intruderSpeed      <= 1200

property7 : Bool
property7 = forall x . largeVerticalSeparation x =>
  not (advises strongLeft x) and not (advises strongRight x)
```

Property 8 - states that for a large vertical separation and if the previous advisory was "weak left", the network will either output COC or continue advising "weak left".

```
largeVerticalSeparationAndPreviousWeakLeft : InputVector -> Bool
largeVerticalSeparationAndPreviousWeakLeft x =
  0    <= x ! distanceToIntruder <= 60760    and
  -pi  <= x ! angleToIntruder    <= -0.75*pi and
  -0.1 <= x ! intruderHeading    <= 0.1      and
  600  <= x ! speed              <= 1200     and
  600  <= x ! intruderSpeed      <= 1200

property8 : Bool
property8 = forall x . largeVerticalSeparationAndPreviousWeakLeft
  x =>
  (advises clearOfConflict x) or (advises weakLeft x)
```

Property 9 - even if the previous advisory was "weak right", the presence of a nearby intruder will cause the network to output a "strong left"advisory instead.

```
previousWeakRightAndNearbyIntruder : InputVector -> Bool
previousWeakRightAndNearbyIntruder x =
  2000 <= x ! distanceToIntruder <= 7000       and
  -0.4 <= x ! angleToIntruder    <= -0.14      and
  -pi  <= x ! intruderHeading    <= -pi + 0.01 and
  100  <= x ! speed              <= 150        and
  0    <= x ! intruderSpeed      <= 150

property9 : Bool
property9 = forall x . previousWeakRightAndNearbyIntruder x =>
  advises strongLeft x
```

Property 10 - for a far away intruder, the network advises COC.

```
intruderFarAway2 : InputVector -> Bool
intruderFarAway2 x =
  36000 <= x ! distanceToIntruder <= 60760       and
  0.7   <= x ! angleToIntruder     <= pi         and
  -pi   <= x ! intruderHeading     <= -pi + 0.01  and
  900   <= x ! speed               <= 1200       and
  600   <= x ! intruderSpeed       <= 1200

property10 : Bool
property10 = forall x . intruderFarAway2 x
                        => advises clearOfConflict x
```

One can hopefully observe that through our DSL the properties are easily interpretable and akin to how one would write them in a traditional functional programming language. Not only this, but the tool will type check the specification to ensure no type errors are introduced in design phase (something quite difficult to do across 43 separate property files). Furthermore, since our DSL is dependently typed so we can specify the dimensions of the tensor, as well as the type of data stored within it. This means that it impossible to mess up indexing into tensors, e.g. if you changed `distanceToIntruder = 0` to `distanceToIntruder = 5` the specification would fail to type-check.

B Autonomous Vehicle Controller: Staying on the Road

In our final scenario an autonomous vehicle is travelling along a straight road of width 6 parallel to the x-axis, with a varying cross-wind that blows perpendicular to the x-axis. The vehicle has an imperfect sensor that it can use to get a (possibly noisy) reading on its position on the y-axis, and can change its velocity on the y-axis in response. The car's controller takes in both the current sensor reading and the previous sensor reading and its goal is to keep the car on the road. The setup is illustrated in Figure 2.

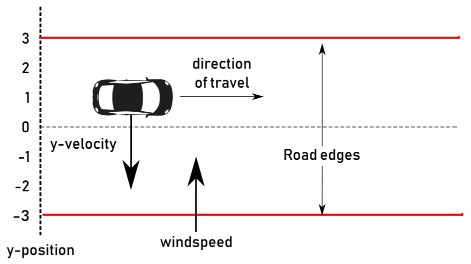

Fig. 2: A simple model of an autonomous vehicle compensating against a cross-wind.

Although not proved here, it turns out that the following specification is sufficient to guarantee that the car will never leave the road:

Initial Setup , we first define the types of the inputs (InputVector) as well as the inputs themselves([currentSensor,previousSensor]), and the network (controller).

```
type InputVector = Tensor Rat [2]

currentSensor  = 0
previousSensor = 1

network controller : InputVector -> Rat
```

Safety Property , then we define the safety property as the controller never steering more than a distance of 3.25 from the correct path.

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

safeOutput : InputVector -> Bool
safeOutput x = -1.25 < controller x + 2 * (x ! currentSensor)
               - (x ! previousSensor) < 1.25

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

From this specification, Vehicle automatically generates the following Marabou queries:

```
x0 >= -3.25                          x0 >= -3.25
x0 <= 3.25                           x0 <= 3.25
x1 >= -3.25                          x1 >= -3.25
x1 <= 3.25                           x1 <= 3.25
-y0 -2.0x0 +x1 >= 1.25               y0 2.0x0 -x1 >= 1.25
```

Although we don't provide details here, on the Github repo there is a full worked example of how this specification may be formally verified and then exported to Agda. In Agda, we then prove that such a controller will never steer the car off the road. Although this is only a toy example, we hope it demonstrates the potential of Vehicle,