



## Vehicle Tutorial Chapter 1: Getting Started

Ekaterina Komendantskaya and Matthew Daggitt (today's presentors), on behalf of the Vehicle team

# In this chapter...



We will:

- ▶ ... introduce Main building blocks of Vehicle as a Programming Language

# In this chapter...



We will:

- ▶ ... introduce Main building blocks of Vehicle as a Programming Language
- ▶ ... get to practice working with Vehicle

# In this chapter...



We will:

- ▶ ... introduce Main building blocks of Vehicle as a Programming Language
- ▶ ... get to practice working with Vehicle
- ▶ ... use the famous ACAS Xu benchmark to show Vehicle's work flow – from specification to verification

# In this chapter...



We will:

- ▶ ... introduce Main building blocks of Vehicle as a Programming Language
- ▶ ... get to practice working with Vehicle
- ▶ ... use the famous ACAS Xu benchmark to show Vehicle's work flow – from specification to verification
- ▶ ... identify PL problems (cf Introduction) that are resolved by Vehicle

Recap: four PL problems



## Recap: four PL problems



- $I^O$  Interoperability – properties are not portable between training/counter-example search/ verification.
- $I^P$  Interpretability – code is not easy to understand.
- $I^J$  Integration – properties of networks cannot be linked to larger control system properties.
- $E^G$  Embedding gap – little support for translation between problem space (as in original spec) and input space (at neural network level).

## Recap: ACAS Xu



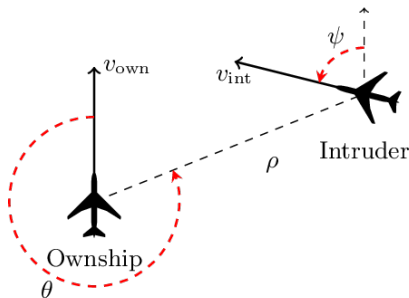
A collision avoidance system for unmanned autonomous aircraft.

Inputs:

- ▶ Distance to intruder,  $\rho$
- ▶ Angle to intruder,  $\theta$
- ▶ Intruder heading,  $\varphi$
- ▶ Speed,  $v_{own}$
- ▶ Intruder speed,  $v_{int}$

Outputs:

- ▶ Clear of conflict
- ▶ Strong left
- ▶ Weak left
- ▶ Weak right
- ▶ Strong right







### Definition (ACAS Xu: Property 3)

*If the intruder is directly ahead and is moving towards the ownship, the score for COC will not be minimal.*

$$\begin{aligned} &1500 \leq \rho \leq 1800 \wedge -0.06 \leq \theta \leq 0.06 \\ &\wedge \psi \geq 3.10 \wedge v_{own} \geq 980 \wedge v_{int} \geq 960 \\ &\Rightarrow \text{the score for COC} \neq 0 \end{aligned}$$

# Table of Contents



Vehicle' Syntax



Let us build the ACAS Xu specification.

We start with types of input and output vectors, as well as types of ACAS Xu networks

```
type InputVector = Vector Rat 5  
type OutputVector = Vector Rat 5
```

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

The Vector type represents a mathematical vector, or in programming terms can be thought of as a fixed-length array.



Types for values are automatically inferred by **Vehicle**. For example, we can declare the number  $\pi$  and its type will be inferred as rational:

```
pi = 3.141592
```

# Working with vectors



- ▶ some input or output pre-processing maybe expected when defining a neural network.

## Example

It is assumed that the ACAS Xu inputs and outputs are normalised, i.e. the network does not work directly with units like  $m/s$ . However, the specifications we want to write should ideally concern the original units.

# Working with vectors



- ▶ some input or output pre-processing maybe expected when defining a neural network.

## Example

It is assumed that the ACAS Xu inputs and outputs are normalised, i.e. the network does not work directly with units like  $m/s$ . However, the specifications we want to write should ideally concern the original units.

- ▶ This is an instance of “problem space / input space mismatch”
- ▶ ... that is very common in neural net verification
- ▶ Being able to reason about problem space (alongside the input space) is a feature that distinguishes **Vehicle** from majority of the mainstream neural network verifiers

# Vector normalisation



For clarity, we define a new type synonym for unnormalised input vectors which are in the problem space.

```
type UnnormalisedInputVector = Vector Rat 5
```

Next we define the range of the inputs that the network is designed to work over.

```
minimumInputValues : UnnormalisedInputVector  
minimumInputValues = [0,0,0,0,0]
```

```
maximumInputValues : UnnormalisedInputVector  
maximumInputValues = [60261.0, 2*pi, 2*pi, 1100.0, 1200.0]
```

```
meanScalingValues : UnnormalisedInputVector  
meanScalingValues = [19791.091, 0.0, 0.0, 650.0, 600.0]
```

# Vector manipulation



An alternative method to vector definition is to use the ‘foreach’ constructor, which is used to provide a value for each ‘index i’.

```
minimumInputValues : UnnormalisedInputVector  
minimumInputValues = foreach i . 0
```

Let us see how ‘foreach’ works with vector indexing.  
We can now define the normalisation function that takes an input vector and returns the unnormalised version.

```
normalise : UnnormalisedInputVector -> InputVector  
normalise x = foreach i .  
    (x ! i - meanScalingValues ! i) / (maximumInputValues ! i)
```



# Vector manipulation



An alternative method to vector definition is to use the ‘foreach’ constructor, which is used to provide a value for each ‘index i’.

```
minimumInputValues : UnnormalisedInputVector  
minimumInputValues = foreach i . 0
```

Let us see how ‘foreach’ works with vector indexing.  
We can now define the normalisation function that takes an input vector and returns the unnormalised version.

```
normalise : UnnormalisedInputVector -> InputVector  
normalise x = foreach i .  
    (x ! i - meanScalingValues ! i) / (maximumInputValues ! i)  
... our first acquaintance with functions!
```

# Functions and types



`<name> : <type>`

`<name> [<args>] = <expr>`

Functions make up the backbone of the **Vehicle** language.

# Functions and types



```
<name> : <type>  
<name> [<args>] = <expr>
```

Functions make up the backbone of the **Vehicle** language.

```
validInput : UnnormalisedInputVector -> Bool  
validInput x = forall i .  
    minimumInputValues ! i <= x ! i <= maximumInputValues ! i
```

# Functions and types



```
<name> : <type>  
<name> [<args>] = <expr>
```

Functions make up the backbone of the **Vehicle** language.

```
validInput : UnnormalisedInputVector -> Bool  
validInput x = forall i .  
  minimumInputValues ! i <= x ! i <= maximumInputValues ! i
```

Our first acquaintance with predicates and quantifiers!

One of the main advantages of **Vehicle** is that it can be used to state and prove specifications that describe the network's behaviour over an infinite set of values.



## Function Composition: Exercise

What are the types of functions ‘acasXu’ and ‘normalise’:

```
normAcasXu : UnnormalisedInputVector -> OutputVector  
normAcasXu x = acasXu (normalise x)
```

# Pre-defined functions and predicates



We have already used:

\*  
/  
!  
<=

## Exercise

What do they stand for?

# Lets verify ACAS Xu!



```
distanceToIntruder = 0    -- measured in metres
angleToIntruder    = 1    -- measured in radians
intruderHeading    = 2    -- measured in radians
speed              = 3    -- measured in metres/second
intruderSpeed      = 4    -- measured in meters/second
```

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

The fact that all vector types come annotated with their size means that it is impossible to mess up indexing into vectors, e.g. if you changed ‘distanceToIntruder = 0’ to ‘distanceToIntruder = 5’ the specification would fail to type-check.

## Property 3



**If the intruder is directly ahead and is moving towards the ownship, the score for COC will not be minimal.**



## Property 3



If the intruder is directly ahead and is moving towards the ownship, the score for COC will not be minimal.

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

## Property 3



If the intruder is directly ahead and is moving towards the ownship, the score for COC will not be minimal.

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

### Exercise!

1. Can you identify whether the specification is written in terms of input space or problem space? How do you know?
2. Can you spot another pre-defined **Vehicle** function? What is it?

## Property 3



**If the intruder is directly ahead and is moving towards the ownship, the score for COC will not be minimal.**

## Property 3



If the intruder is directly ahead and is moving towards the ownship, the score for COC will not be minimal.

```
movingTowards : UnnormalisedInputVector -> Bool
```

```
movingTowards x =
```

```
  x ! intruderHeading >= 3.10  and
```

```
  x ! speed           >= 980    and
```

```
  x ! intruderSpeed   >= 960
```

## Property 3



If the intruder is directly ahead and is moving towards the ownship, the score for **COC** will not be minimal.

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

### Exercise!

1. Can you spot one more pre-defined **Vehicle** function? What is it?

There is little left to do!



**If the intruder is directly ahead and is moving towards the ownship, the score for COC will not be minimal.**

# There is little left to do!



If the intruder is directly ahead and is moving towards the ownship, the score for COC will not be minimal.

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

# There is little left to do!



If the intruder is directly ahead and is moving towards the ownship, the score for COC will not be minimal.

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

## Exercise!

1. Can you guess the purpose of the syntax

```
@property
```

?

2. What kind of domain 'forall' ranges over? Is it finite or infinite?



# How to run Vehicle



## Checklist

1. a verifier installed (Marabou);
2. the actual network is supplied in an ONNX format
3. **Vehicle** is installed.

# How to run Vehicle



## Checklist

1. a verifier installed (Marabou);
2. the actual network is supplied in an ONNX format
3. **Vehicle** is installed.

```
vehicle verify \  
  --specification acasXu.vcl \  
  --verifier Marabou \  
  --network acasXu:acasXu_1_7.onnx \  
  --property property3
```

Verifying properties:

```
property3 [=====] 1/1 queries  
result: counterexample found  
x: [1799.9886669999978, 1.9509286320000003e-2,  
    3.09999732192, 980.0, 1017.6036]
```

# Exercise 1 (moderate): $\epsilon$ -ball Robustness



```
type Image = Tensor Rat [28, 28]
type Label = Index 10
validImage : Image -> Bool
validImage x = forall i j . 0 <= x ! i ! j <= 1

@network
classifier : Image -> Vector Rat 10

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

@parameter
epsilon : Rat

boundedByEpsilon : Image -> Bool
boundedByEpsilon x = forall i j . -epsilon <= x ! i ! j <= epsilon

robustAround : Image -> Label -> Bool
robustAround image label = forall perturbation .
  let perturbedImage = image - perturbation in
    boundedByEpsilon perturbation and validImage perturbedImage =>
      advises perturbedImage label

@dataset
trainingImages : Vector Image n

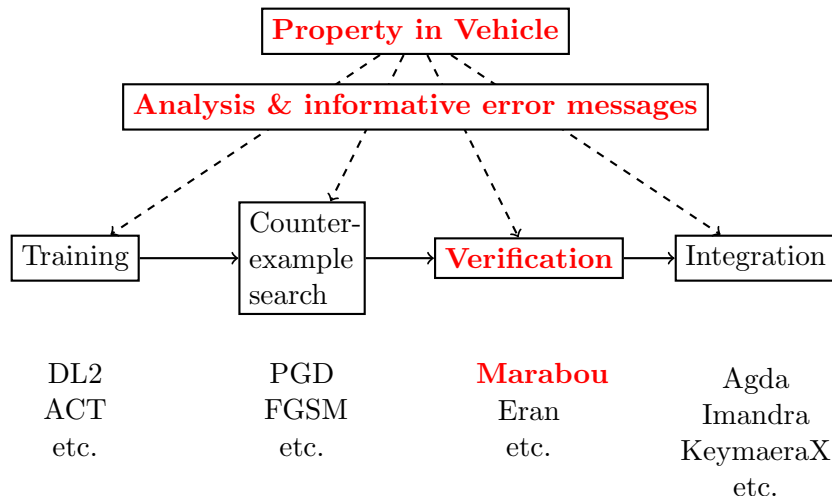
@dataset
trainingLabels : Vector Label n

@property
robust : Vector Bool n
robust = foreach i . robustAround (trainingImages ! i) (trainingLabels ! i)
```

## Vehicle ...



the part that we have seen



# Concluding Exercise



Which of the four PL problems we addressed?

- $I^O$  Interoperability – properties are not portable between training/counter-example search/ verification.
- $I^P$  Interpretability – code is not easy to understand.
- $I^f$  Integration – properties of networks cannot be linked to larger control system properties.
- $E^G$  Embedding gap – little support for translation between problem space (as in original spec) and input space (at neural network level).

## Exercise 2 (hard): ACAS Xu: Property 1



In the previous lecture, we introduced ACAS Xu property 1.

- ▶ Can you formalise that property in Vehicle?
- ▶ Can you spot the instance of the embedding gap, this time concerning the network's output?