IT UNIVERSITY OF COPENHAGEN

Updating the Vehicle-lang Python bindings to support training with constraints

Research Project
Computer Science Department
IT University of Copenhagen

submitted by
Gusts Gustavs Grīnbergs BSc
Course of study: Computer Science
on December 4, 2025

Supervisor: Assoc. Prof. Alessandro Bruni

Abstract

Vehicle is a research platform that unifies formal specification and differentiable machine learning, yet its Python bindings had drifted from the evolving compiler and could no longer serve as a single artefact for verification and training. This research project tackles that gap by analysing the legacy bindings, identifying the missing coverage for constructs such as quantified searches, and redesigning the stack so the compiler's JSON output can be consumed directly by modern TensorFlow and PyTorch workflows. I revamp an internal AST decoder, a consolidated abstraction layer for builtins and samplers, a refreshed TensorFlow backend, and introduce a new PyTorch backend that both share one translation pipeline and optional dependency guardrails. Pytest evidence shows the updated loaders ingest the golden benchmark suite, the backends emit matching declaration contexts, and the generated losses participate in constraint-only and mixed-objective optimisation without bespoke rewrites, while documenting the intentionally skipped bounds-only fixtures and disabled long-running demonstrations that remain future work.

ableofcontents

Chapter 1

Introduction

Deep neural networks have achieved impressive performance across domains such as computer vision and natural language processing, yet they remain surprisingly brittle. Small, carefully chosen perturbations to correctly classified inputs can cause state-of-the-art models to fail with high confidence [8, 6]. This phenomenon of adversarial examples has motivated a large body of work on robustness and formal verification of neural networks, particularly for safety-critical applications where such failures are unacceptable.

In parallel, formal methods researchers have developed increasingly powerful verification techniques for neural networks and other machine learning components. Tools based on satisfiability modulo theories, abstract interpretation, and specialized solvers such as Reluplex have demonstrated that it is feasible to prove non-trivial safety properties of neural controllers in restricted settings. However, most existing tools are tailored to specific architectures or input domains, and are typically applied á posteriori, i.e. after a model has been trained. This separation between training and verification can lead to an expensive iterate-and-fix cycle, and makes it difficult to guarantee that the model actually used in deployment satisfies the same specification that was verified [7, 2].

Property-driven machine learning has emerged as a promising response to this challenge. Instead of viewing formal specifications as something checked only after training, property-driven approaches integrate logical constraints directly into the learning process. Differentiable logics allow one to encode arbitrary first-order properties as loss terms that guide optimisation, while adversarial training-style methods can approximate worst-case violations within regions of interest. Recent work by Flinkow et al. [5] proposes a general framework for property-driven training that unifies these ideas: logical specifications are compiled into differentiable objectives, and training is steered towards models that satisfy both data-driven and specification-driven requirements.

The Vehicle project [3] takes a complementary perspective focused on the specification and verification side. Vehicle is a domain-specific language and toolchain for specifying properties of neural and neuro-symbolic programs once, then compiling them to multiple back-ends, including SMT solvers, interactive theorem provers, and specialised neural network verifiers. Vehicle aims to bridge the “em-

bedding gap" between machine learning frameworks and formal verification tools by providing a common verification condition language, together with type-safe compilation paths to different verification back-ends.

Despite this progress, there remains a gap between property-driven training frameworks and general-purpose verification condition languages such as Vehicle. On the one hand, frameworks like ANTONIO [1] demonstrate how domain-specific benchmarks and pipelines can be built to make verification of NLP models practical, by carefully shaping datasets and network architectures to match the capabilities of existing verifiers. On the other hand, Vehicle focuses on specifying properties in an abstract, solver-independent way, but its Python bindings have historically been geared more towards offline verification and experimentation than towards being directly usable as training-time constraints in modern deep learning workflows.

This research project addresses this gap by updating and extending the Python bindings of Vehicle so that the same formally specified property can be used consistently for both training and verification. Concretely, it realigns the Python-side abstract syntax tree with the current Vehicle compiler output, streamlines and internalises the abstraction layer that mediates between Vehicle programs and differentiable back-ends, and provides coherent TensorFlow and PyTorch backends that interpret Vehicle's loss logic as differentiable loss functions. The work also separates the command-line compilation utilities from the loss machinery, thereby avoiding unnecessary ML-framework imports for verification-only workflows while still offering optional TensorFlow and PyTorch extras for property-driven training. On top of this architecture, the project delivers a high-level Python API that allows practitioners to load a Vehicle specification, obtain loss functions corresponding to named properties, and integrate them directly into standard training loops.

Research Question

The central question of this research project is:

> **How can Vehicle's existing Python bindings be extended so that they correctly reflect the current compiler and loss logic, and support quantifiers and their samplers for use in both training and verification?**

This question is motivated by the observation that, while the overall structure of the Python bindings and JSON AST parsing for Vehicle already existed, changes in the core language (in particular around quantified search and loss constructs) had introduced a mismatch between the compiler and its Python interface. In collaboration with the core Vehicle developers, this research project therefore focuses on (i) realigning the Python AST and abstraction layer with the current compiler, (ii) co-designing and implementing the representation of `SearchRatTensor` nodes and the corresponding sampler interfaces needed for property-driven training [5],

and (iii) demonstrating that, once these pieces are in place, the same formally specified properties can be exercised both as differentiable losses in TensorFlow/PyTorch and as verification conditions compiled to existing back-ends. The resulting architecture contributes an incremental but necessary step towards workflows in which specification, training, and verification remain technically and semantically aligned.

Chapter 2

Background

Before going over the specifics of the code changes made as part of this research project, it is useful to review some background material on Vehicle, its DSL, and the general approach to property-driven training using differentiable logics.

Once these concepts have been introduced, this chapter will survey related work in the areas of formal verification of neural networks, property-driven machine learning, and existing tools that aim to bridge the gap between specification, training, and verification.

2.1 The Vehicle Project

The Vehicle project [3] is a domain-specific language (DSL) and toolchain for specifying and verifying properties of neural and neuro-symbolic programs. Vehicle allows users to write formal specifications in a high-level, solver-independent language, which can then be compiled to multiple back-ends, including SMT solvers, interactive theorem provers, and specialized neural network verifiers.

Vehicle's core idea is to provide a common verification condition language that abstracts away the details of specific verification tools. This allows users to write properties once and then target different back-ends depending on their needs. Vehicle also includes type-safe compilation paths to ensure that the generated verification conditions are well-formed and compatible with the chosen back-end. In the implementation described later, these responsibilities are exposed through lightweight `vehicle_lang.compile` utilities in the Python bindings that shell out to the compiler without importing any training-specific dependencies, reserving the heavier loss functionality for a separate module. The compiler can also be invoked directly from the command line using the `vehicle` CLI tool.

2.2 Vehicle's Domain-Specific Language

Vehicle's DSL is designed to express a wide range of properties relevant to neural and neuro-symbolic programs. The language includes constructs for defining variables, functions, and logical properties, as well as specialized constructs for dealing

with neural network components. Vehicle supports first-order logic with quantifiers, allowing users to express properties that involve universal or existential quantification over inputs or internal states. Vehicle specifications are typically written in a structured format, with declarations for variables and functions, followed by property declarations that specify the desired behaviors or constraints.

Specifically, references to external resources such as datasets or neural network models can be made within Vehicle programs via decorators. This allows properties to be expressed in terms of real-world data and model behaviors, making the specifications more relevant and applicable to practical scenarios and reusable with different models and specification parameters.

For example, a simple specification might be the standard robustness property for image classifiers, stating that small perturbations to an input image should not change the predicted class. The Vehicle repository already includes such an example, `examples/mnist-robustness.vcl`, which the remainder of this section dissects to illustrate the structure and features of Vehicle's DSL.

Type declarations and network interface. We begin by declaring the tensor shapes for images and labels, followed by the neural network we plan to reason about. Vehicle treats neural networks as black boxes annotated with `@network`, so the specification only needs their type signatures. Vehicle supports networks saved as ONNX files, but the specification itself remains agnostic to the underlying implementation. However, the downstream training and verification back-ends have individual constraints on which architectures are supported [11].

```
type Image = Tensor Real [28, 28]
type Label = Index 10

@network
classifier : Image -> Tensor Real [10]
```

Listing 2.1: Types and network interface used by the MNIST property

Characterising admissible perturbations. Next we write helper predicates that say which tensors are valid images and which perturbations are allowed (bounded by `epsilon`). The `advises` helper captures the semantics of the classifier's prediction, and `robustAround` lifts these ingredients into the familiar "if every bounded perturbation is still valid, then the classifier keeps the label" statement.

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

@parameter
epsilon : Real

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

```
9
10 advises : Image -> Label -> Bool
11 advises x y = forall j . j != y => classifier x ! y > classifier x
       ↪ ! j
12
13 robustAround : Image -> Label -> Bool
14 robustAround image label = forall perturbation .
15   let perturbed = image - perturbation in
16   boundedByEpsilon perturbation and validImage perturbed =>
17     advises perturbed label
```

Listing 2.2: Reusable predicates describing validity and robustness

Quantifying over the dataset. Finally we quantify the property over the training set. Vehicle's `@parameter(infer=True)` and `@dataset` decorators let the specification refer to data supplied at compile time. The `@property` block below states that, for every training index `i`, the classifier is robust around that sample.

```
1  @parameter(infer=True)
2  n : Nat
3
4  @dataset
5  trainingImages : Vector Image n
6
7  @dataset
8  trainingLabels : Vector Label n
9
10 @property
11 mnistRobust : Vector Bool n
12 mnistRobust = foreach i .
13   robustAround (trainingImages ! i) (trainingLabels ! i)
```

Listing 2.3: Dataset-aware property used for training and verification

This specification can then be compiled to different back-ends for verification, or interpreted as differentiable loss functions for property-driven training, as described in the next section. The updated Python bindings realise this duality by hosting all differentiable-loss logic inside the `vehicle_lang.loss` package, which loads the AST through the new `loss._ast` module and instantiates backend-specific translations only when the TensorFlow or PyTorch optional extras are present.

2.3 Differentiable Logics

Differentiable logics (DLs) bridge the gap between symbolic reasoning and continuous optimization, providing a mechanism to integrate logical specifications directly into the training of machine learning models [9]. The core principle is the translation of discrete logical properties—which are typically non-differentiable—into

smooth, differentiable loss functions. This enables the use of standard gradient-based optimization algorithms (such as stochastic gradient descent) to train models that not only fit empirical data but also respect formal correctness properties.

2.3.1 From Logic to Loss

The translation process maps Boolean logic semantics onto a continuous domain (typically $[0, 1]$ or $\mathbb{R}$), effectively "fuzzifying" the logic. This is achieved through the use of *t-norms* (triangular norms) and their associated conorms, which provide differentiable approximations for logical connectives:

- **Conjunction ($\wedge$):** Often approximated using the minimum function (Gödel t-norm), product (Product t-norm), or the Łukasiewicz t-norm ($\max(0, x + y - 1)$).
- **Disjunction ($\vee$):** Approximated via the maximum function or probabilistic sums.
- **Implication ($\rightarrow$):** Derived from the chosen t-norm residuals, allowing properties like "if input is A, then output must be B" to be encoded as a penalty term.

While propositional connectives are straightforward to map, first-order quantifiers pose a significant challenge. Universally quantified properties (e.g., "for all inputs x in region R, condition $P(x)$ holds") theoretically require infinite checks. Traditional DL approaches approximate these quantifiers using Monte Carlo sampling, calculating the average or maximum loss over a batch of random points. However, naive sampling often fails to detect sparse counter-examples, leading to models that satisfy properties on average but fail in worst-case scenarios [4].

2.3.2 Property-Driven Training Framework

To address the limitations of sampling-based DLs, recent work by Flinkow et al. [5] proposes a general framework for *property-driven training*. This approach compiles high-level logical specifications into differentiable objectives that explicitly target the model's worst-case behaviors.

Crucially, this framework introduces a mechanism for approximating the worst-case violation of properties within specified regions of interest, often defined as hyper-rectangles in the input space. Rather than relying solely on random sampling, the training process integrates an adversarial search (or "attack") step that actively seeks inputs maximizing the logical loss. This results in a robust training objective:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{data}} + \lambda \cdot \max_{x \in S} \mathcal{L}_{\text{logic}}(x) \tag{2.1}$$

where S represents the region defined by the specification. By optimizing against these "most violating" inputs, the model is forced to learn a decision boundary that

is robust to perturbations and consistent with the formal specification, effectively combining the benefits of adversarial training with the expressivity of first-order logic.

The implementation of this framework (integrated into the Vehicle ecosystem) allows users to express arbitrary first-order properties which are then automatically compiled into these hybrid loss functions, utilizing optimizers capable of handling the inner maximization problem efficiently.

2.4 Related Work

The integration of symbolic reasoning into machine learning pipelines has emerged as a critical area of research, driven largely by the need for safety and reliability of neural networks. This section outlines the progression from adversarial robustness and formal verification to the recent developments in differentiable logics and neuro-symbolic frameworks.

2.4.1 Adversarial Robustness and Explanability

The susceptibility of deep neural networks to adversarial perturbations was famously highlighted by Szegedy et al. [8], who demonstrated that imperceptible changes to input data could lead to high-confidence misclassifications. Goodfellow et al. [6] further formalized this via the generation of adversarial examples, arguing that the linearity of high-dimensional models was a primary cause of this brittleness. These foundational works established the necessity for models that do not merely fit training distributions but satisfy formal robustness properties.

2.4.2 Formal Verification of Neural Networks

To address these reliability concerns, the formal methods community developed techniques to prove properties of neural networks post-training. Early breakthroughs include *Reluplex* [7], which extended the Simplex algorithm to handle non-convex ReLU activation functions, enabling the verification of safety properties in collision avoidance systems.

More recently, Casadio et al. [2] proposed a principled methodology for evaluating robustness as a formal verification property. They highlighted the distinction between empirical robustness (resistance to known attacks) and verifiable robustness (mathematical guarantees), a gap that motivates the need for better training objectives. Similarly, in the domain of Natural Language Processing (NLP), the *ANTONIO* framework [1] demonstrated how systematic benchmarking can be used to assess verification tools against formal specifications.

2.4.3 Differentiable Logics and Training

While verification acts as a gatekeeper, it does not inherently improve the model during the learning phase. Differentiable Logics (DLs) attempt to bridge this gap by translating logical constraints into differentiable loss functions. Fischer et al. introduced *DL2* [4], a system that allows users to query networks with logic and train them to satisfy constraints by fuzzifying boolean connectives.

However, the effectiveness of DLs relies heavily on the quality of the translation from logic to arithmetic. Van Krieken et al. [9] provided a comprehensive analysis of various t-norms (fuzzy logic operators), showing that the choice of operator significantly impacts the optimization landscape and the gradient quality during neuro-symbolic learning.

Chapter 3

Approach

In this chapter I describe how I aligned and extended Vehicle's Python bindings so that they correctly reflect the current Vehicle compiler and loss logic, and how I co-designed the representation and sampling semantics of search-based loss constructs such as `SearchRatTensor`. The focus is on the concrete engineering decisions that were necessary to make Vehicle specifications usable as training-time constraints in TensorFlow and PyTorch, while preserving Vehicle as the single source of truth for the logical properties.

3.1 Overview of the Existing Design

Before making changes, I analysed the state of the Python bindings on the `dev` branch. At a high level, three conceptual layers were already present:

1. A JSON AST mirror and loader in `vehicle_lang.ast`, which called the Vehicle compiler with `--jsoncompileloss` and decoded the result into Python dataclasses. However, this module was significantly out of date with respect to the current Vehicle AST.

2. An abstraction layer in `compile.abc` that defined backend-agnostic interfaces for builtins and translations from Vehicle programs to some target representation.

3. A Python translation in `compile.python` that consumed the AST and produced Python `ast.Module`s, which were then compiled and executed into a `declaration_context` dictionary. This dictionary provided concrete implementations of low-level operations in TensorFlow.

Since the existing structure at a high level was sound, just out of date, and missing key components for quantifiers, I decided to build on top of it where possible.

The approach of this research project is therefore not to invent a new architecture from scratch, but to realign these layers with the current Vehicle compiler,

clarify and clean-up existing abstractions, implement quantifier parsing, design an interface for samplers, provide a default sampler, and improve the overall code quality and coherence of the bindings.

3.2 AST Ingestion from the Vehicle Compiler

The first step was to bring Python-side AST ingestion back in sync with the Vehicle compiler, addressing the staleness of the JSON AST mirror described above. In collaboration with the core Vehicle developers, I introduced an updated and refactored internal module `vehicle_lang.loss._ast` to replace the outdated public `vehicle_lang.ast` in the compilation pipeline.

3.2.1 Internal AST Module

The `vehicle_lang.loss._ast` module provides:

- A `load` function that invokes the Vehicle compiler with the appropriate `--jsoncompileloss` and `--logic` options, for a given specification file and list of declarations.
- A set of `_nodes` dataclasses that mirror the current JSON AST produced by the Vehicle compiler, including constructs for quantifiers and operations in the new tensor representation.
- A refactored JSON decoder `_decode.py` that uses modern Python type annotations and improved error handling, making it easier to evolve alongside the compiler.

By prefixing a module or script with an underscore, we signal that it is internal and not part of the public API. This gives us freedom to evolve the AST representation as needed without breaking user code as well as improving the usability of the codebase [10].

3.3 Abstraction Layer

The second step was to streamline the abstractions that enable the modularity of the framework. On `dev`, the responsible `compile.abc` module was public and included overlapping concepts such as `Builtins` and `ABCBuiltins`. This made it difficult both to understand and to change. Another issue was that the usage of generic types was inconsistent, specifically, `vcl.type_name` was used in multiple places, but the specific module that was imported as `vcl` varied. This led to confusion about which types were actually being used in different contexts.

3.3.1 Internal Abstract Base Class Package

On my branch I introduced the new internal module `vehicle_lang.loss._abc` which refines and consolidates this abstraction into four main components:

`_types.py` Defines generic type variables for indices, rational numbers, tensors and for the program/declaration/expression types that a translation operates on. This makes the interfaces parametric in the backend, while remaining tied to the structure of the Vehicle AST.

`_builtins.py` Defines a single `ABCBuiltins[Index,Rat,Tensor]` interface with a coherent set of abstract methods corresponding to Vehicle's core tensor operations, reductions and dimension handling (e.g. `RatTensor`, `ReduceAddRatTensor`, dimension constructors, and tensor constructors such as `ConstTensor` and `StackTensor`). This interface serves as the contract that backend implementations (e.g. TensorFlow, PyTorch) must satisfy in order to support Vehicle's loss logic.

`_samplers.py` Introduces an `ABCSampler[Index,Tensor]` interface that defines what API a sampler should provide for quantifiers. Its central method `get_loss` returns a tensor-valued loss that realises the semantics of a search region. A key design choice of the maintainers is to support a wide range of approaches, since the sampler can return 1 or more loss values that are combined with a compiled reduction operation (depending on the logic and quantifier). This allows for strategies such as random sampling, quasi-random sampling, or adversarial attacks to be implemented using the same interface.

`_translation.py` Defines a generic `ABCTranslation[Program,Declaration,Expression]` that pattern-matches over the new `_ast._nodes` types. It has concrete methods for translating programs and declarations, while leaving expression translation abstract. This design allows backends to inherit from `ABCTranslation` and only implement the expression-level translation logic, while reusing the program/declaration-level logic.

This modular design makes it easy to add more backends in the future if they can satisfy these interfaces.

3.3.2 Separating ITP, Queries, and Loss Logic

After stabilising the AST and abstraction layers I revisited the overall package organisation. On `dev`, the public `vehicle_lang.compile` module addressed every downstream consumer simultaneously: it invoked the compiler for ITP exports, SMT-style query formats, and the differentiable loss path. Consequently, importing `vehicle_lang` indiscriminately pulled in TensorFlow even when a user only produced, for example, VNNLib queries. I therefore separated the responsibilities along their natural boundaries:

- **Compilation and verification targets** remain under `vehicle_lang.compile` (and neighbouring modules) and expose thin wrappers around the CLI (e.g. `compile_specification` and `call_vehicle`). These modules depend solely on the shared typing layer and session helpers, which keeps them lightweight.
- **Training-time loss logic** was relocated to a dedicated `vehicle_lang.loss` package. This package owns AST decoding, abstract base classes, backend translations, and backend-specific helpers. Because the key modules live under `_ast`, `_abc`, `_tensorflow`, and `_pytorch`, importing `vehicle_lang` no longer instantiates TensorFlow or PyTorch unless a loss backend is explicitly requested.
- **Optional dependencies** are guarded via the shared `require_optional_dependency` helper. TensorFlow and PyTorch are imported only inside the modules that require them, and the resulting error message directs the user to the corresponding `pipinstall"vehicle_lang[tensorflow]"` or `[pytorch]` extra when missing.

This separation makes the relationship between the verification and training pipelines explicit while minimising unnecessary transitive dependencies: command-line workflows retain a minimal footprint, whereas researchers who require the loss module opt into the relevant extras and obtain the full backend stack.

3.4 Testing Methodology

To assess whether these architectural changes satisfy the research question, I relied on the existing Pytest infrastructure maintained by the Vehicle authors and extended it with three additional groups of tests that target the major technical risks identified during this project:

- **AST synchronisation**: golden Vehicle specifications (both synthetic unit tests and the production reachability/monotonicity/wind-controller cases) are compiled with `--jsoncompileloss` and loaded via `vehicle_lang.loss._ast.load`. These tests guard against drift between the compiler and the Python mirror whenever new constructs such as search tensors or quantifiers are introduced.
- **Backend translation**: backend-specific suites instantiate the TensorFlow and PyTorch translations via `loss._common.load_loss_specification`, exercise the `ABCBuiltins` and `ABCSampler` implementations, and ensure that optional dependencies are only imported when the corresponding backend module (e.g. `vehicle_lang.loss.tensorflow` or `vehicle_lang.loss.pytorch`) is referenced.
- **Training integration**: gradient-based experiments use the generated loss functions inside TensorFlow and PyTorch optimisers, confirming that the

compiled losses remain differentiable, that samplers can be injected per quantified variable, and that constraint-only as well as mixed task/constraint objectives converge.

These additions complement the pre-existing regression suites and all follow the same Pytest conventions (including `pytest.importorskip` for optional dependencies). Verification-focused tests therefore remain runnable without TensorFlow or PyTorch installed, while the new training- and backend-specific suites activate only when the relevant extras are available.

3.5 Sampler Design and Implementation

A central part of my contribution lies in the design and implementation of how `SearchRatTensor` nodes and their associated sampling behaviour are represented on the Python side. This was done in close collaboration with the Vehicle maintainers.

3.5.1 Requirements from Property-Driven Training

From the perspective of property-driven machine learning [5], in the case of the `forall` quantifier, `SearchRatTensor` captures a search over a region of the input space (described by lower and upper bounds) returning the loss evaluated on worst-case violations of a specification (best-case for the `exists` quantifier). For training, the sampler needs to:

- Find points within the specified search region.
- Evaluate the loss objective at those points.

To fulfil these requirements, especially with strategies like adversarial attacks which require gradient information, we had to provide more information than just the bounds and loss function to the sampler. Therefore, we designed the sampler interface to accept:

1. The dimensions and their bounds.
2. A representation of the loss objective as a callable function.
3. A flag indicating whether to minimise or maximise the loss. This is needed because different logics may have different absolute truth and absolute falsity values, specifically, some logics have $[[true]]_l < [[false]]_l$.

3.5.2 Sampler Interface and Implementation

Based on these principles, we designed the `ABCSampler` interface in `_samplers.py`.

Each backend provides a more specific abstract class that inherits from `ABCSampler` and specifies the concrete types involved (e.g. `tf.Tensor` or `torch.Tensor`). Additionally, each backend provides a default implementation called `[Pytorch,Tensorflow]DefaultSampler` that uses a fast gradient sign method to find adversarial examples [6]. This default sampler is sufficient for many use cases and serves as a reference implementation since the step size is inferred from the bounds, specifically ensuring that the bounds of the search region can be reac

If a user wants to use a custom sampling strategy, they can implement their own class conforming to `ABCSampler` and define a dictionary which maps the name of the variable being searched for to their corresponding sampler instance. This allows for having separate samplers for different quantified variables in the same specification, e.g. one for image perturbations and another for sensor noise.

However, we do allow for ducktyping here: as long as the user-provided sampler has a `get_loss` method with the correct signature, it will be accepted at runtime. This design choice was made to lower the barrier for users to provide custom samplers without needing to formally inherit from the abstract base class.

3.6 Extending and Aligning the translations

With the abstractions and sampler design in place, the next step was to extend and align the `PythonTranslation` class to implement all of the necessary methods of `ABCTranslation`, and to do the same for both TensorFlow and the new PyTorch backend so that they implement `ABCBuiltins` and `ABCSampler` correctly.

3.6.1 Python translation

The `PythonTranslation` class now resides in `vehicle_lang.loss._python` and defines how Vehicle AST nodes are translated into Python `ast` nodes. A helper in `loss._common` instantiates this translation via a backend-specific factory, ensuring that every backend reuses the same traversal and substitutes only its `builtins` and `samplers`. Low-level operations are delegated to those backend implementations through the runtime dictionaries.

In the case of the potentially user provided samplers, the availability of the specific sampler instance is checked at runtime by looking up the variable name in a `samplers` dictionary that is passed to the compiled loss function. If no sampler is defined for this variable, then a `KeyError` is returned. This is an intentional design choice to ensure that users are aware when they have not provided a sampler for a quantified variable, instead of defaulting to a sampler which may not be appropriate for their use case.

The translation's `compile` method generates Python bytecode which is executed with a declaration context that includes the builtins and samplers, thereby adding an entry to the provided dictionary containing the generated callable loss

function. The method terminates by returning this updated dictionary containing the result of the execution. As this path is now invoked solely through `loss._common.load_loss_specification`, none of this machinery runs when a user relies exclusively on the verification-oriented `compile` modules.

3.6.2 TensorFlow Backend

The TensorFlow backend was updated to implement the new `ABCBuiltins` and `ABCSampler` interfaces:

- Each Vehicle builtin is mapped to an appropriate TensorFlow operation or composition of operations.
- While the translation of the `SearchRatTensor` node is handled by `PythonTranslation`, the specific sampler interface and default implementation is defined in `tensorflow.samplers.py`.

Most of the needed operations already existed in the original implementation, so the changes were mostly around updating method signatures and adding missing operations to align with the new abstractions. The module now imports TensorFlow through the shared `require_optional_dependency` helper, so users who never call `vehicle_lang.loss.tensorflow.load_specification` can still install the package without TensorFlow present.

3.6.3 PyTorch Backend

On the `dev` branch there was no PyTorch. On my branch I implemented a parallel backend that conforms to `ABCBuiltins` and `ABCSampler` for `torch.Tensor`:

- Tensor operations and reductions mirror those of the TensorFlow backend, but are implemented using PyTorch's API.
- The sampler uses PyTorch tensors and operations to instantiate search regions and compute approximate worst-case losses for `SearchRatTensor`.

Since the abstractions were designed to be backend-agnostic, this implementation was straightforward, requiring mostly one-to-one mappings of operations from TensorFlow to PyTorch with minor differences in API conventions.

As with TensorFlow, PyTorch is treated as an optional dependency. The backend code is only imported when a caller uses `vehicle_lang.loss.pytorch`, and the helper explains which `pip` extra to install if the module is missing. This keeps the default installation lean while still supporting both ecosystems.

This ensures that the same Vehicle specification can be compiled into PyTorch-based loss functions, enabling property-driven training in PyTorch with no changes to the original specification. This was a key goal of the project, as it broadens the applicability of Vehicle to a wider range of projects. This also serves as a step towards integrating the work of *Flinkow et al*[5] within Vehicle.

3.7 High-Level Python API and Package Design

Finally, I integrated these internal changes into a high-level Python API aimed at end users. Each backend exposes a dedicated `load_specification` helper (e.g. `vehicle_lang.loss.tensorflow.load_specification` or `vehicle_lang.loss.pytorch.load_specification`) that wraps the shared loader in `loss._common`:

- The caller passes a Vehicle specification file, optionally restricts the declaration set, selects a differentiable logic, and can provide custom samplers.
- Internally, it calls `_ast.load` to obtain the updated AST, and then uses the appropriate backend-specific translation to populate a user-provided `declaration_context` mapping.
- Because the backend module is selected explicitly, the optional dependency is loaded only when needed. The resulting context contains callable loss functions for each named declaration (property) in the Vehicle specification, ready to be integrated into standard training loops.

The package exposes only the necessary components for users, while keeping the internal modules such as `_ast`, `_abc`, and backend implementations private. While a user can still access these internal modules if needed, the high-level API is designed to cover the common use cases without requiring direct interaction with the lower-level details. This also serves to make it easier to understand which parts of the codebase are stable and intended for public use versus those that are more experimental or subject to change.

This design keeps the internal complexity of AST decoding, abstraction, and sampler semantics hidden from users, while still allowing advanced users to override or extend sampler behaviour when needed. It also reiterates the separation principle introduced earlier: importing `vehicle_lang` for CLI compilation remains lightweight, and only those who adopt property-driven training incur the TensorFlow or PyTorch dependencies.

Chapter 4

Evaluation and Analysis

This chapter evaluates whether the updated Python bindings satisfy the research question, namely that the same Vehicle specification can be exercised consistently for both verification and training. The evidence is drawn from the Pytest suites in `vehicle-python/tests`, which cover AST ingestion, backend translation, and differentiable training across TensorFlow and PyTorch.

4.1 AST Synchronisation Results

The `test_lossdl2_load.py` and `test_golden_specs.py` suites compile every synthetic specification in `tests/data` as well as the large "golden" benchmarks (reachability, monotonicity, wind controller) via `vehicle --json compile loss`. The resulting JSON programs are decoded through `vehicle_lang.loss._ast.load` with `DifferentiableLogic.DL2`. All files load successfully except for the two bounds-only quantifier fixtures that the suite intentionally skips because of a known compiler bug.[1] Passing these tests demonstrates that the private `_ast` module mirrors the current compiler, including the constructs introduced for `SearchRatTensor` and quantified search, thereby eliminating the original drift between the compiler and the Python bindings.

4.2 Backend Validation

4.2.1 TensorFlow

The suite in `test_lossdl2_exec.py` loads each specification through `vehicle_lang.loss.tensorflow.load_specification`, exercises arithmetic primitives, tensor constructors, and network calls, and validates the returned callables either through scalar expectations or through custom validators (e.g. the "bounded" property that injects a dummy sampler). Additional golden tests compile the large specifications to TensorFlow and confirm that user-facing declarations are

[1] Vehicle issue #1004

emitted. Together, these tests demonstrate that the `TensorFlowTranslation`, `TensorFlowBuiltins`, and sampler implementations satisfy the `ABCBuiltins`/`ABCSampler` contracts and expose the expected declaration context without requiring any additional configuration beyond optional sampler dictionaries.

4.2.2 PyTorch

PyTorch coverage is provided by `test_pytorch_backend.py` and `test_pytorch_integration.py`. The backend tests exercise every tensor primitive (creation, arithmetic, reductions, dimension constructors) and verify CUDA compatibility when available. The integration tests compile real specifications via the Vehicle CLI, feed the JSON program to `PyTorchTranslation`, and compare the resulting symbol tables against those produced by the TensorFlow translation, ensuring that both backends expose identical user declarations. Golden-spec compilation further confirms that production-scale specifications execute in the PyTorch backend. Passing these tests demonstrates feature parity between the two backends and shows that the backend-agnostic abstractions introduced in Chapter 3 operate as intended.

4.3 Training Integration Experiments

The training-focused evidence comes from `test_training.py`, which instantiates both TensorFlow and PyTorch models and optimisers. Two "constraint-only" tests initialise networks that violate the `output_bounded` property generated from `test_trainable.vcl`. After 50 optimisation steps using only the compiled loss, both frameworks reduce the loss value and lower the maximum output, demonstrating that the generated constraint is differentiable and can steer the model without any task loss. The combined-loss tests create a simple regression task ($y = 2x$) and optimise a weighted sum of task loss and constraint loss. They verify that initial losses are finite, gradients are finite (a critical property given the use of FGSM-style samplers), and that a single optimisation step reduces the combined objective. A multi-step PyTorch test further shows monotonic improvement over 20 iterations. Collectively, these experiments confirm that the generated losses integrate seamlessly with automatic differentiation and that sampler-provided perturbations do not break gradient flow.

4.4 Optional Dependencies and Disabled Scenarios

Every test module uses `pytest.importorskip` to enforce the optional-dependency policy: verification and AST tests run without TensorFlow or PyTorch installed, while backend and training suites activate only when the corresponding extras are available. This behaviour demonstrates that the separation described in Chapter 3

is effective in practice. Two longer-running end-to-end demonstrations (bounded TensorFlow training and MNIST robustness) remain guarded by `ifFalse` blocks to avoid prohibitive runtime in continuous integration; the necessary code paths are nevertheless covered indirectly through the regression and training suites, and they provide a blueprint for future extended experiments.

4.5 Interpretation and Remaining Risks

The evaluation shows that all critical invariants of the research question hold: (i) AST ingestion remains aligned with the compiler, (ii) both backends implement the loss logic with feature parity and optional dependency isolation, and (iii) the resulting loss functions are differentiable and effective inside standard training loops. The remaining risks are limited to the bounds-only quantifier fixtures that stay intentionally disabled until the compiler bug tracked in Vehicle issue #1004 is fixed, and the disabled full end-to-end demonstrations. Addressing those will require more work on the Vehicle compiler side as well as longer-running experiments.

Chapter 5

Conclusion

The project brought Vehicle's Python bindings closer to serving as a single artefact for both verification and differentiable training. The refreshed modules ingest the current compiler output for the benchmark set, generate backend-neutral loss programs, and execute them in TensorFlow and PyTorch for the covered scenarios, while acknowledging that long-running demonstrations and a handful of compiler edge cases remain open. This chapter distils the inferences, enumerates the concrete contributions, and outlines the research steps still required.

5.1 Conclusion and Discussion

1. The research question is answered for the evaluated benchmarks: specifications compiled with `vehicle --json compile loss` run unchanged in both verification pipelines and TensorFlow/PyTorch training loops (Sections 4–3). The new bindings remove the previously observed drift between the compiler and Python mirror for these cases, while two bounds-only fixtures remain deliberately disabled until the upstream compiler bug is resolved.

2. Backend parity is demonstrated by instantiating the shared `vehicle_lang.loss` translation with TensorFlow and PyTorch builtins (Section 4). Pytest evidence shows that the two backends expose matching declaration contexts across the golden specifications, indicating that framework choice no longer forces specification rewrites.

3. Optional-dependency isolation now matches downstream needs: verification tasks import only the lightweight CLI helpers, while training workflows opt into TensorFlow/PyTorch extras on demand. This separation addresses the installation pain reported at project start and clarifies the boundary between proof-oriented and learning-oriented tooling, albeit without repackaging the legacy modules.

4. Generated losses are usable beyond toy cases, but only within short-running experiments. Constraint-only and mixed-objective tests (§4) show that DL2-based losses and FGSM-style samplers integrate with standard optimisers,

while the longer training demonstrations remain disabled pending further performance work.

5.2 Summary of Contributions

1. Delivered `vehicle_lang.loss._ast`, an internal AST decoder that mirrors the current compiler JSON (including `SearchRatTensor`) and isolates future schema evolution from the public API.

2. Consolidated the abstraction layer inside `vehicle_lang.loss._abc`, unifying type variables, builtin contracts, sampler interfaces, and translation scaffolding so that each backend implements a single coherent contract.

3. Refurbished the TensorFlow backend and introduced a PyTorch counterpart that reuse the common Python translation; both are positioned as beta-quality implementations pending further optimisation and larger-scale trials.

4. Helped design the sampler interface and made FGSM-inspired samplers for both frameworks, enabling quantified searches to yield usable losses during training while allowing users to plug in custom strategies.

5. Extended the Pytest suite to cover AST synchronisation, backend translation, and short training integration runs, providing regression protection for the abstractions even though CI still skips the longest demonstrations.

5.3 Limitations and Future Research

1. Address the two bounds-only quantifier fixtures that remain deliberately skipped in `test_lossdl2_load.py` because of the known compiler bug; unblocking them would close the last known gap in AST coverage.

2. Re-enable the long-running end-to-end demonstrations (bounded TensorFlow training and MNIST robustness) under a dedicated CI job or performance budget. Their reintroduction would provide empirical validation for larger models and datasets.

3. Explore richer sampler strategies (e.g. quasi-random lattices, certified adversaries, or gradient-free search) that plug into `ABCSampler` while preserving differentiability. Comparative studies could quantify how sampler choice affects convergence and robustness.

4. Generalise the backend abstractions to additional targets, such as JAX or ONNX-compatible runtimes, to further reduce the effort required to bring Vehicle constraints into diverse ML stacks.

5. Investigate tighter integration with property-driven training frameworks (e.g. automated curriculum generation or joint optimisation with task loss schedulers), turning the compiled Vehicle losses into first-class citizens inside experiment management tooling.

Bibliography

[1] Marco Casadio, Luca Arnaboldi, Matthew L Daggitt, Omri Isac, Tanvi Dinkar, Daniel Kienitz, Verena Rieser, and Ekaterina Komendantskaya. ANTONIO: Towards a systematic method for generating nlp benchmarks for verification. In *Proceedings of the 6th Workshop on Formal Methods for ML-Enabled Autonomous Systems (FoMLAS)*, 2023.

[2] Marco Casadio, Ekaterina Komendantskaya, Matthew L. Daggitt, Wen Kokke, Guy Katz, Guy Amir, and Idan Refaeli. Neural network robustness as a verification property: A principled case study. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification*, pages 219–231, Cham, 2022. Springer International Publishing.

[3] Matthew L Daggitt, Wen Kokke, Robert Atkey, Ekaterina Komendantskaya, Natalia Slusarz, and Luca Arnaboldi. Vehicle: Bridging the embedding gap in the verification of neuro-symbolic programs. In *Proceedings of the 10th International Conference on Formal Structures for Computation and Deduction (FSCD)*, LIPIcs, 2025.

[4] Marc Fischer, Mislav Balunovic, Dana Drachsler-Cohen, Timon Gehr, Ce Zhang, and Martin Vechev. DL2: Training and querying neural networks with logic. In *International Conference on Machine Learning*, pages 1931–1941. PMLR, 2019.

[5] Thomas Flinkow, Marco Casadio, Colin Kessler, Rosemary Monahan, and Ekaterina Komendantskaya. A general framework for property-driven machine learning. *arXiv preprint arXiv:2505.00466*, 2025.

[6] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2015.

[7] Guy Katz, Clark Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, pages 97–117, Cham, 2017. Springer International Publishing.

[8] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2014.

[9] Emile van Krieken, Evgenia Acar, and Frank Geura. Analyzing and improving differentiable logic for neuro-symbolic learning. *arXiv preprint arXiv:2201.10099*, 2022.

[10] Guido van Rossum, Barry Warsaw, and Nick Coghlan. Style guide for Python code. PEP 8, Python Software Foundation, 2001.

[11] Haoze Wu, Omri Isac, Aleksandar Zeljić, Teruhiro Tagomori, Matthew Daggitt, Wen Kokke, Idan Refaeli, Guy Amir, Kyle Julian, Shahaf Bassan, Pei Huang, Ori Lahav, Min Wu, Min Zhang, Ekaterina Komendantskaya, Guy Katz, and Clark Barrett. Marabou 2.0: A versatile formal analyzer of neural networks, 2024.