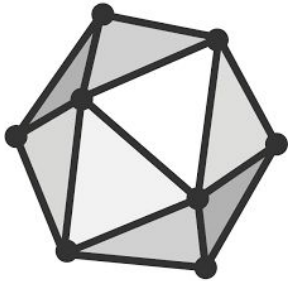
Formal Safety Verification of ONNX Operators

Using Frama-C/WP plug-in

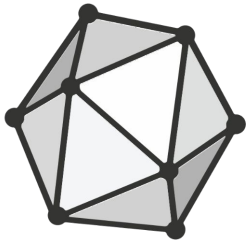


Carolina Sá pg 61453 | Miguel Liquito pg 61480 | Gonçalo Caixeiro pg 60260

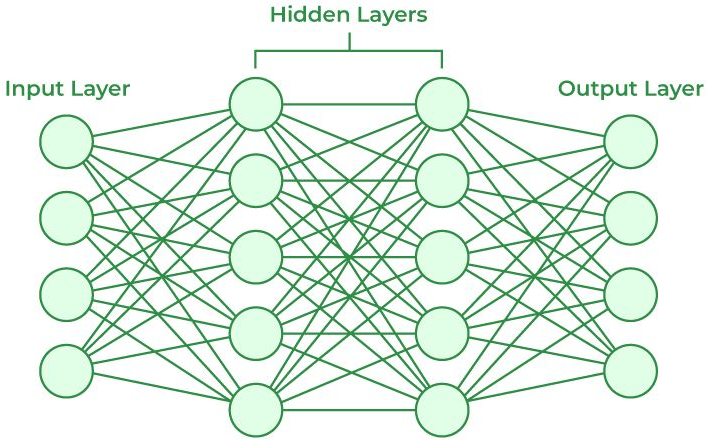
Advisors: Maria João Frade, Jorge Sousa Pinto

Class: Project in Formal Methods of Programming

Introduction

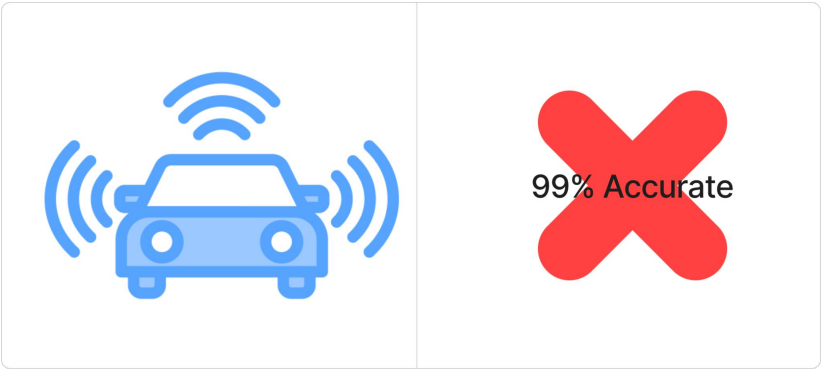


ONNX



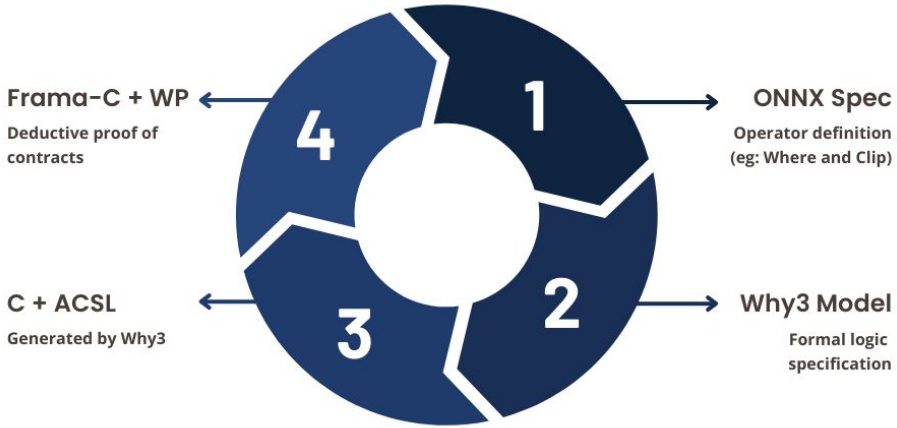
Can We Trust a Neural Network?

- Deep learning is deployed in safety-critical systems: autonomous vehicles, clinical diagnostics, robotics
- NN inference is implemented in C/C++ - subject to memory bugs like any other C code
- A single out-of-bounds write or shape mismatch can cause catastrophic hardware failure
- Training proves nothing about the safety of the inference implementation
- **Goa**l: Apply formal deductive verification to the C code that runs neural network inference



Context

- ONNX - standard format for ML models, describes a neural network as a graph
- Tensor - multi-dimensional array of numbers (Scalar 0D, Vector 1D, Matrix 2D…)
- Operators - nodes of the ONNX graph, take tensors as input and output altered tensors

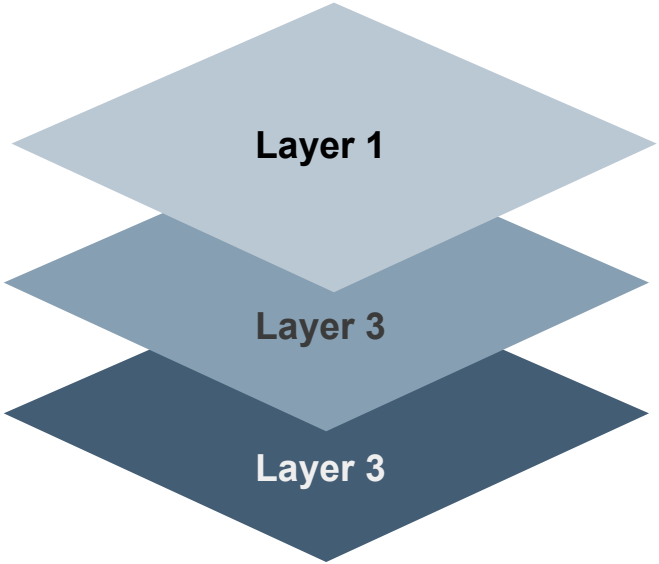


Safety in a Critical environment written in C.

In order to assure that a C code base does not break during production, the following requirements must be met:

- No out-of-bounds memory access
- No integer overflow or undefined arithmetic
- No null pointer references
- All loop invariants written **must** hold during verification.

Three Layers of Verification



- **Why3 Formal Specification:** Mathematical definition of what each operator must do
- **C Implementation:** Generated from Why3; uses pointers, loops, heap memory
- **Frama-C ACSL Annotations:** Bridge between the two formal contracts that must respect the Why3 specification and prove safety in C

- **Rule**: No case forbidden by Why3 may be allowed by Frama-C
- **Double Check:** Why3 + Verification of generated C code

Recap

Done: 

- Tensor Library specification
- Clip & Where preconditions and postconditions
- Clip & Where Functional Verification
- Clip & Where Chain Specification

Incomplete: 

- MatMul Specification
- Full LeNet-5 chain proof (this goal had not yet been set at that time)
- Specify the operators used in the Lenet network.

Goals

1. MatMul functional correctness - prove $Y[i][j] = \Sigma A[i][k] \cdot B[k][j]$
2. Complete postconditions for all 6 operators (conv, reshape, maxpool, add, identity, relu,)
3. Prove the full LeNet - 5 inference chain end-to-end
    a. Generalise shapes - parametric, not hardcoded 28×28
    b. ONNX-compatible operators - add the operators' attributes.

MatMul: Functional Correctness

New: recursive dot product logic (mirrors Why3 spec):

```
/*@
  logic real dot_product_logic(double *a_data, integer a_cols,
                               double *b_data, integer b_cols,
                               integer row, integer col,
                               integer i, integer iter) =
    i >= iter ? 0.0 :
      a_data[row * a_cols + i] * b_data[i * b_cols + col]
      + dot_product_logic(a_data, a_cols, b_data, b_cols,
                          row, col, i + 1, iter);
*/
```

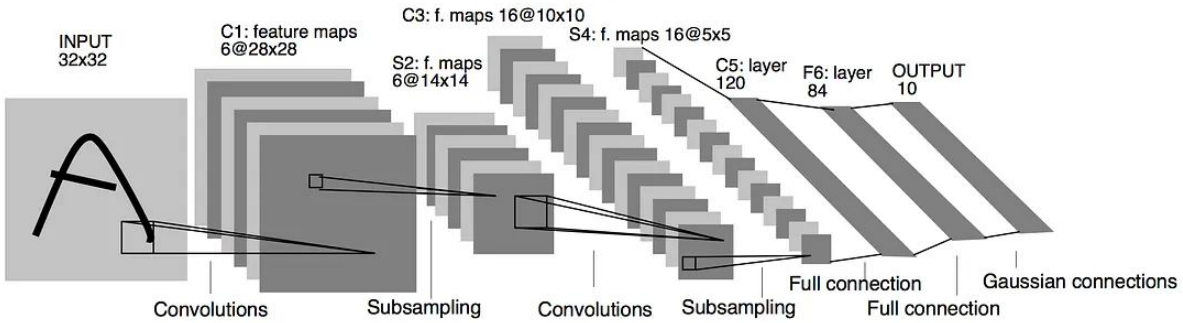
New postcondition:

```
  ensures \result == 1 ==>
    \forall integer i, j;
      0 <= i < a.t_dims[0] && 0 <= j < b.t_dims[1] ==>
        r.t_data[i * r.t_dims[1] + j] ==
          dot_product_logic(a.t_data, a.t_dims[1],
                            b.t_data, b.t_dims[1],
                            i, j, 0, a.t_dims[1]);
*/
```

What is LeNet-5?

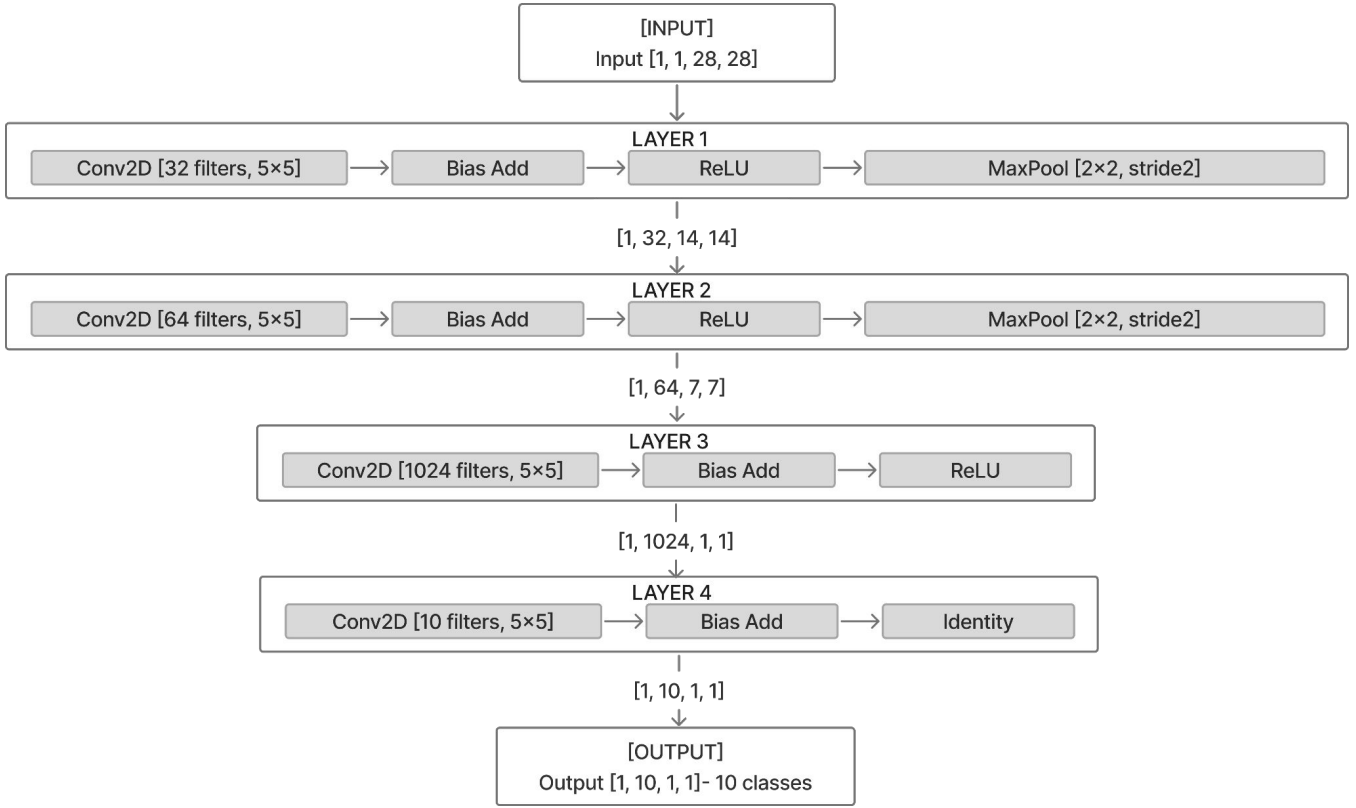
LeNet-5 - The Target Network

- Classic CNN by LeCun et al. (1998), designed for handwritten digit recognition (MNIST)
- 4 computational blocks: Conv + Pool + Conv + Pool + Conv + FC
- Input: 1x1x28x28 image tensor | Output: 1x10x1x1 class scores
- Exported as an ONNX model - a standard open format for neural networks
- Small enough to be formally verifiable; representative of the CNN family



LeNet-5 example

LeNet-5 Architecture



Proving the Full LeNet Chain

Strategy: Modular Verification via Stubs

- Each operator has full ACSL contracts but no implementation body
- lenet_forward calls them in sequence
- Frama-C WP proves every transition is safe using the contracts alone

What the chain proves:

- No out-of-bounds memory access - anywhere in the full forward pass
- No aliasing between the 27 intermediate buffers (\separated)
- Shapes propagate correctly layer by layer

Design decision - Memory Parameterization:

All 27 intermediate buffers are passed as arguments (not malloc inside) → Frama-C WP cannot track heap allocations in deep call chains

Completing Operator Contracts

Postconditions added for every operator:

Operator	What was proved
ReLu	`Y[i] = max(0, X[i])` for all elements
Identity	`Y[i] = X[i]` for all elements
Reshape	`Y[k] = X[k]` → data unchanged; rank matches shape tensor
Conv	Output spatial dims follow ONNX formula
MaxPool	Output dims follow full parametric formula
Add	Output dims match broadcast rules

ONNX-Compliant Reshape

Problem at M2: Reshape was a simplified stub (ignored allowzero and -1 in shape tensor S)

Solution - new function signature:

```
void ctensor_reshape(struct ctensor x, struct ctensor s, int32_t allowzero,
                     struct ctensor out);
```

New constraints:

- allowzero ∈ {0, 1} - ONNX attribute
- At most one -1 in S —-auto-infer that dimension
- If S[i] = 0 and allowzero = 0, copy dimension from X
- Total element count of X must equal that of out

Generalised Shapes: conv_out_dim

From hardcoded constants → axiomatic formula:

```
logic integer conv_out_dim(integer in_d, integer pad_b, integer pad_e,
                           integer dilation, integer kernel, integer stride) =
  (in_d + pad_b + pad_e - (dilation * (kernel - 1) + 1)) / stride + 1;
```

- Used for both Conv and MaxPool shape derivation
- MaxPool = conv_out_dim with dilation = 1
- Proof now covers any valid input shape, not just 28×28

MaxPool before → after:

Version 1	Version 2
`out = in / 2`	Full formula with kernel, stride, pad, dilation

Results

325 / 325 Proof Obligations - 100% Proved

Solver	Goals	Share
Qed (internal simplifier)	269	83%
Alt-Ergo 2.6.2	59	16%
CVC5 1.1.2	4	1%
Timeouts/Failures	0	-

- Average time per goal: < 1 second
- Frama-C exit code: 0 (clean success)

```
[wp] Proved goals:  325 / 325
  Qed:             269 (0.52ms-79ms-1.1s)
  Alt-Ergo 2.6.2:   52 (20ms-556ms-1.2s)
  CVC5 1.1.2:        4 (108ms-717ms)
```

What the Proof Guarantees

- **Absolute Memory Safety** No out-of-bounds reads/writes. No null pointer dereferences.
- **No Pointer Aliasing** All 27 intermediate buffers are in disjoint memory regions - proved with \separated.
- **Shape Consistency** Tensor dimensions propagate correctly through every layer per ONNX formulas.
- **Functional Semantics** (selected operators)

  - ReLU: `Y[i] = max(0, X[i])`
  - Reshape / Identity: data unchanged, `Y[k] = X[k]`
  - MatMul: `Y[i][j] = Σ A[i][k]·B[k][j]`

Technical Challenges

Challenge	Solution
`malloc` unverifiable in deep chains	Memory Parameterization - 27 buffers as arguments
Integer division hard for SMT solvers	`DivisionProperties` axiomatic block
Context too large → timeout risk	`\separated` split into 5 groups + intermediate asserts
MatMul loop induction step	Ghost function `dp_step_proof`

Limitations: WP plugin

8. Conclusion

> Voilà, c'est fini ...
>
> *Jean-Louis Aubert, Bleu Blanc Vert, 1989*

... for this introduction to the proof of C programs using Frama-C and WP.

Along this tutorial, we have seen how we can use these tools to specify what we expect of our programs and verify that the source code we have produced indeed corresponds to the specification we have provided. This specification is provided using annotations of our functions that includes the contract they must respect. These contracts are properties required about the input to ensure that the function will correctly work, which is specified by properties about the output of the function and enforced by the tool that allow us to check specific problems related to the use of C (namely, the absence of runtime errors).

Starting from specified programs, WP is able to produce the weakest precondition of our functions, provided what we want in postcondition, and to ask some provers if the specified precondition is compatible with the computed one. The reasoning is completely modular, which allows proving functions in isolation from each other and to compose the results.

WP cannot currently work with dynamic allocation. A function that would use it could not be proved. However, even without dynamic allocation, a lot of function can be proved since they work with data-structures that are already allocated. And these functions can then be called with the certainty that they perform a correct job. If we cannot or do not want to prove the client code of a function, we can still write something like this:

```
/*@
  requires some_properties_on(a);
  requires some_other_on(b);

  assigns ...
  ensures ...
*/
void my_function(int* a, int b){
  //this is indeed the "assert" defined in "assert.h"
  assert(/*properties on a*/ && "must respect properties on a");
  assert(/*properties on b*/ && "must respect properties on b");
}
```

Which allows us to benefit from the robustness of our function having the possibility to debug an incorrect call in a source code that we cannot or do not want to prove.

Writing specifications is sometimes long or tedious. Higher-level features of ACSL (predicates, logic functions, axioms) allow us to lighten this work, as well as our programming languages allow us to define types containing other types, functions calling functions, bringing us to the

```
[rte:annot] annotating function ctensor_where
[wp] 43 goals scheduled
[wp] Proved goals: 43 / 43
Qed: 31 (6ms-8ms-68ms)
Alt-Ergo 2.4.3: 12 (22ms-327ms-565ms)
```

```
[wp] 176 goals scheduled
[wp] [Timeout] typed_cdim_create_1_assigns_normal_part3 (Qed 2ms) (Alt-Ergo) (Stronger, 3 warnings)
[wp] [Timeout] typed_cdim_create_1_assert_rte_mem_access (Qed 2ms) (Alt-Ergo) (Stronger, 3 warnings)
[wp] [Timeout] typed_cdim_create_1_assigns_normal_part1 (Qed 2ms) (Alt-Ergo) (Stronger, 3 warnings)
[wp] [Timeout] typed_cdim_create_2_assigns_normal_part3 (Qed 2ms) (Alt-Ergo) (Stronger, 3 warnings)
[wp] [Timeout] typed_cdim_create_2_assert_rte_mem_access (Qed 2ms) (Alt-Ergo) (Stronger, 3 warnings)
[wp] [Timeout] typed_cdim_create_2_assert_rte_mem_access_2 (Qed 3ms) (Alt-Ergo) (Stronger, 3 warnings)
[wp] [Timeout] typed_cdim_create_2_assigns_normal_part4 (Qed 8ms) (Alt-Ergo) (Stronger, 3 warnings)
[wp] [Timeout] typed_cdim_create_2_assigns_normal_part1 (Qed 1ms) (Alt-Ergo) (Stronger, 3 warnings)
[wp] Proved goals: 174 / 182
Terminating: 3
Unreachable: 3
Qed: 96 (1ms-20ms-303ms)
Alt-Ergo 2.4.3: 72 (8ms-295ms-8.8s)
Timeout: 8
```

Conclusions: Why3-Frama C patterns

Key Patterns	Description
Data Representation	Mapping **Why3 algebraic records and coordinate list types to flat 1D heap-allocated double pointer structures (struct ctensor)** and flattened index arithmetic **(i * cols + j).**
Implicit vs. Explicit Separation	Transitioning from implicit value semantics (immutability in Why3) to explicit **\separated** contracts in ACSL to prevent alias-induced proof failures.
Loop Invariants & Memory State	Elevating loop invariants from simple functional correctness to carrying the full memory validation state **(\valid, \valid_read)** at every loop iteration.
Inductive Witnesses	Translating Why3 recursive lemmas into ACSL recursive logic functions paired with ghost C function wrappers to act as induction proofs for the SMT solvers.

Established a systematic, reusable method to map why3 representation to C code

Conclusions:

Compositionality & Full Network Verification

Key Points	Description
Proof of Operator Compositionality (Chaining)	Demonstrated that the **post-conditions (ensures)** of **layer N** mathematically and structurally satisfy the **preconditions (requires)** of **layer N+1.**
Modular Verification Scale	Enabled verification of a pipeline **without re-analyzing operator implementations in context**, preventing state-space explosion.
End-to-End LeNet-5 Inference Chain	Verified the entire generalized neural network pipeline (comprising **Convolution, ReLU, MaxPool, Reshape**, and Fully Connected layers).
Automated Proof	Discharged all 325 verification goals using the Frama-C WP plugin backed by SMT solvers (Alt-Ergo/Z3).

Bridges the gap between single-operator safety and the verification of complete, deep learning inference chains.
325 Verification Goals: 100% Discharged.

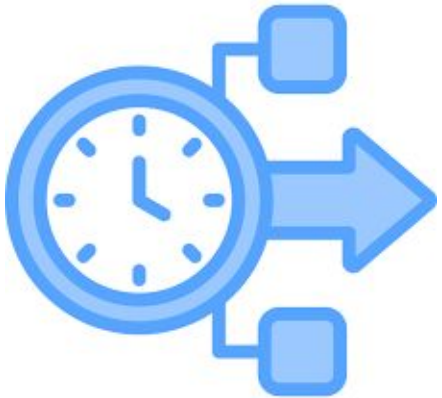
Conclusions

- Formal ACSL contract library for 9 ONNX operators
- Verified tensor support library (ctensor, cdim_size, coffset)
- 100% proof of memory safety and functional correctness for Clip, Where, MatMul
- Proof of operator compositionality (chaining)
- 100% proof of the full generalized LeNet-5 inference chain (325 goals)
- Reusable Why3-to-Frama-C translation patterns for ONNX operators

frama C Software Analyzers

Future Work

- Arithmetic correctness of convolutions (element-wise)
- Floating-point rounding analysis (Gappa / Fluctuat)
- More ONNX operators: Softmax, BatchNorm
- Larger networks: ResNet, MobileNet



Thank You!