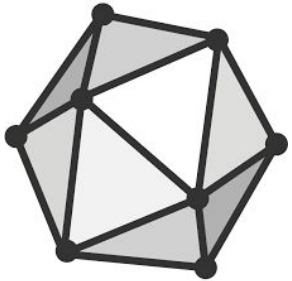
Formal Safety Verification of ONNX Operators

Using Frama-C/WP plug-in



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

Advisors: Maria João Frade

Class: Project in Formal Methods of Programming

Recap of Milestone 1

- **Achieved**: Memory Safety for `where` and `clip` operators.
- **Verified**:
  - No out-of-bounds accesses.
  - No integer overflows.
  - No null pointer dereferences.
- **Limitations**:
  - WP timeouts with dynamic allocation (`malloc`).
  - Missing proof of mathematical accuracy.

```
gojalo@Caixeiro:~/mei/pmfp/proj/UM-FramaC/clip/code$ frama-c -wp -wp-
sor_clear ctensor.c cindex.c
[kernel] Parsing ctensor.c (with preprocessing)
[kernel] Parsing cindex.c (with preprocessing)
[wp] Warning: Missing RTE guards
[wp] 23 goals scheduled
[wp] Proved goals:   23 / 23
  Qed:              19 (2ms-1ms-13ms)
  Alt-Ergo 2.6.2:    3 (28ms-99ms-192ms)
  Z3 4.8.12:         1 (58ms)
```

Output Example

Goals

- **Target**: Functional verification.
- **Objective**:
  - Prove the C code precisely mirrors the Why3 mathematical specifications, respecting the scope and allowed or prohibited cases
  - Express the functionality of the operators with ACSL and verify it with Frama-C.
  - Validate operator chaining, eg: clip(where(x,b))
- **Challenges**:
  - Expressing abstract mathematics in concrete ACSL.
  - Proving complex tensor loops without Prover timeouts.

Functional Verification in `clip`

- **Context**: `clip(x, min, max)`
- **Refining Contracts**:
  - **Memory Shape**: Mapped Why3's `valid_tensor` to C `\valid`.
  - **Scalars**: Ensured `min` and `max` limits behave as strictly bounded 1D scalars.
  - **Matching Dimensions**: Ensured output matrix strictly matches the input matrix dimensions.

```
// — shape compatibility (Why3: tensor x ~= tensor r) ————————————
requires x.t_rank == r.t_rank;
requires \forall integer j; 0 <= j < r.t_rank ==> x.t_dims[j] == r.t_dims[j];

// — scalars for l and m (Why3: is_scalar_tensor l / m) ————————————
requires l.t_rank == 1 && l.t_dims[0] == 1;
requires m.t_rank == 1 && m.t_dims[0] == 1;

// — memory array read validity ————————————————————————————————
requires \valid_read(x.t_data + (0 .. dim_size_logic(r.t_dims, r.t_rank) - 1));
requires \valid_read(l.t_data + (0 .. 0));
requires \valid_read(m.t_data + (0 .. 0));

// — functional postcondition (ITE equality, mirrors loop body) ————
ensures \forall integer i; 0 <= i < dim_size_logic(r.t_dims, r.t_rank) ==>
  r.t_data[i] ==
    (l.t_data[0] <= m.t_data[0]
      ? (x.t_data[i] < l.t_data[0]  ? l.t_data[0]
          : (x.t_data[i] > m.t_data[0] ? m.t_data[0]
              : x.t_data[i]))
      : m.t_data[0]);
```

Refined Contract

```
r.t_data[i] = (fmin(m_background, (fmax((x.t_data[i]), l_background))));
```

`clip` - Axioms and Lemmas

- **The Problem**: Provers struggle with nested conditions inside tensor loops.
- **The Solution**:
  - ACSL Specification for External Functions
  - ACSL `axiomatic Fmin_Fmax_Spec`.
  - Custom **Lemmas**:
    - `clip_case_below` -> forces `min`.
    - `clip_case_above` -> forces `max`.
    - `clip_case_middle` -> maintains `value`.

```
/*@
  axiomatic Fmin_Fmax_Spec {

    logic double my_fmax(double x, double y) =
      x >= y ? x : y;

    logic double my_fmin(double x, double y) =
      x <= y ? x : y;
  }
*/
```

Axiomatic Fmin_Fmax_Spec

```
/*@
  lemma clip_case_below:
    \forall double v, lo, hi;
      lo <= hi ==> v < lo ==>
        my_fmin(hi, my_fmax(v, lo)) == lo;
```

Lemma clip_case_bellow

`clip` - Axioms and Lemmas

- **The Problem**: Provers struggle with nested conditions inside tensor loops.
- **The Solution**:
  - ACSL Specification for External Functions
  - ACSL `axiomatic Fmin_Fmax_Spec.`
  - Custom **Lemmas**:
    - `clip_case_below` -> forces `min`.
    - `clip_case_above` -> forces `max`.
    - `clip_case_middle` -> maintains `value`.

```
/*@ assigns \result \from x, y;
    behavior ge: assumes x >= y; ensures \result == x;
    behavior lt: assumes x < y;  ensures \result == y;
    complete behaviors; disjoint behaviors;
*/
extern double fmax(double x, double y);

/*@ assigns \result \from x, y;
    behavior le: assumes x <= y; ensures \result == x;
    behavior gt: assumes x > y;  ensures \result == y;
    complete behaviors; disjoint behaviors;
*/
extern double fmin(double x, double y);
```

ACSL Specification for External Functions

`clip` - Invariants and Results

- **Postconditions**: Output formally matches the `fmin`/`fmax` axiomatic models for every index.
- **Loop Invariants**: State is correctly preserved step-by-step `$k < i$`.
- **Prover Reports**: 282 of 298 execution goals successfully proved.
- **Result**: Safe, structurally sound, and mathematically perfect execution!

```
// — functional invariant (ITE equality, mirrors loop body) —
loop invariant \forall integer k; 0 <= k < i ==>
  r.t_data[k] ==
    (l.t_data[0] <= m.t_data[0]
      ? (x.t_data[k] < l.t_data[0]  ? l.t_data[0]
          : (x.t_data[k] > m.t_data[0] ? m.t_data[0]
              : x.t_data[k]))
      : m.t_data[0]);
```

Invariant Snippet

```
[wp] Proved goals:  282 / 298
  Qed:              164  (1ms-19ms-397ms)
  Alt-Ergo 2.4.0:   116  (1ms-85ms-867ms)
  CVC4 1.8:          20  (40ms-59ms-90ms)
  Z3 4.8.6:          32  (10ms-34ms-50ms)
```

Example Report

Functional Verification in `where`

- **Context**: Element-wise selection (`cond[i] > 0 ? a[i] : b[i]`).
- **The 3-Step Proof Strategy**:
  1. **Establishment [F-EST]**: Clean entry point for invariant constraints.
  2. **Preservation [F-PRE]**: Memory isolation (`\separated`) & mid-loop anchor assert.
  3. **Bridging [F-USE]**: Closing the gap perfectly to the `m-1` boundary limit.

Functional Verification in `where`

[F-PRE]

```
/*@
  loop invariant \forall integer k; 0 <= k < i ==>
    ( (((double) 0.0) < cond.t_data[k] ==> r.t_data[k] == a.t_data[k]) &&
      (((double) 0.0) >= cond.t_data[k] ==> r.t_data[k] == b.t_data[k]) );
*/
```

[F-EST]

```
r.t_data[i] = ((double) 0.0) < cond.t_data[i] ? a.t_data[i] : b.t_data[i];

 /*@ assert \forall integer k; 0 <= k <= i ==>
       ( (((double) 0.0) < cond.t_data[k] ==> r.t_data[k] == a.t_data[k]) &&
         (((double) 0.0) >= cond.t_data[k] ==> r.t_data[k] == b.t_data[k]) );
 */
```

[F-USE]

```
/*@ assert \forall integer k;
      0 <= k < dim_size_logic(r.t_dims, r.t_rank) ==>
        ( (((double) 0.0) < cond.t_data[k] ==> r.t_data[k] == a.t_data[k]) &&
          (((double) 0.0) >= cond.t_data[k] ==> r.t_data[k] == b.t_data[k]) );
*/
```

`where` - Results

- **Prover Reports**: 287 of 299 execution goals successfully proved.
- **Complexity Handled**: Loop boundaries and aliasing resolved.
- **Remaining Limitations**: Isolated purely to expected Frama-C `malloc` timeouts.

```
[wp] [Stepout] typed_cdim_create_2_assigns_norm
[wp] Proved goals:  287 / 299
  Terminating:       3
  Unreachable:       3
  Qed:             160 (2ms-22ms-319ms)
  Alt-Ergo 2.4.3:  116 (8ms-277ms-13.9s)
  CVC4 1.8:          3 (40ms-106ms)
  Z3 4.8.12:         2 (48ms)
  Timeout:          12
```

Example of a report

`operator chaining` - Results

- **Chaining is proven by matching contracts:** the **ensures** from operator A must satisfy the **requires** of operator B.
- **In order to test we created a small test** and verified it with Frama-C.
    a. **The Contract Handshake:** The postcondition of where automatically guarantees the memory safety and shape compatibility required by clip .
- **Results**: No timeouts during **requires calls.**

```
void build_where_clip_chain(struct ctensor cond, struct ctensor a, struct ctensor b,
                            struct ctensor tmp, struct ctensor l, struct ctensor m,
                            struct ctensor r) {
    ctensor_where(cond, a, b, tmp);
    ctensor_clip(tmp, l, m, r);
}
```

```
[wp] [Qed] Goal typed_build_where_clip_chain_call_ctensor_where_requires_2 : Valid (7ms)
[wp] [Qed] Goal typed_build_where_clip_chain_call_ctensor_where_requires : Valid (7ms)
[wp] [Qed] Goal typed_build_where_clip_chain_call_ctensor_where_requires_4 : Valid (5ms)
[wp] [Qed] Goal typed_build_where_clip_chain_call_ctensor_where_requires_3 : Valid (5ms)
[wp] [Qed] Goal typed_build_where_clip_chain_call_ctensor_where_requires_6 : Valid (4ms)
[wp] [Qed] Goal typed_build_where_clip_chain_call_ctensor_where_requires_5 : Valid (5ms)
[wp] [Qed] Goal typed_build_where_clip_chain_call_ctensor_where_requires_7 : Valid (6ms)
[wp] [Qed] Goal typed_build_where_clip_chain_call_ctensor_where_requires_9 : Valid (6ms)
[wp] [Alt-Ergo 2.4.0] Goal typed_build_where_clip_chain_call_ctensor_where_requires_8 : Valid
```

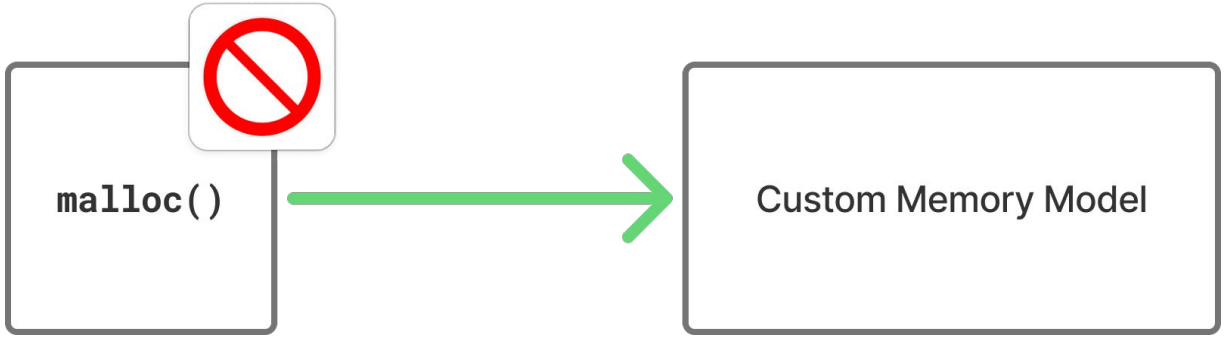
Conclusions

- **Success**: Expanded formal proofs to full **Functional Correctness** for both `where` and `clip`.
- **Complete Pipeline**: Code extracted from Why3 -> proved mathematically accurate in C space.
- **Final Layer Bridged**: We successfully completed the specification verification chain.

$$Y[i] = min(M, max(X[i], L))$$

Future Work

- Handling Dynamic memory (`malloc`) directly via custom external Frama-C memory formalizations.
- Applying this robust functional proof blueprint scaling up to complex operators (`Add`, `MatMul`).



Thank You!