Formal Safety Verification of ONNX Operators

Using Frama-C/WP plug-in

Students:

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

Advisors:

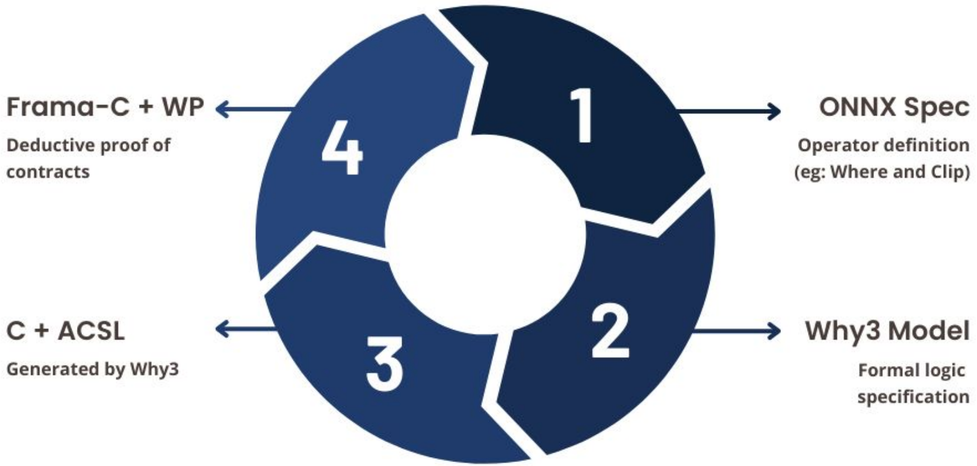
Maria João Frade

Class:

Projeto Métodos Formais de Programação

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



Three Layers.

Each layer tackles a specific area:

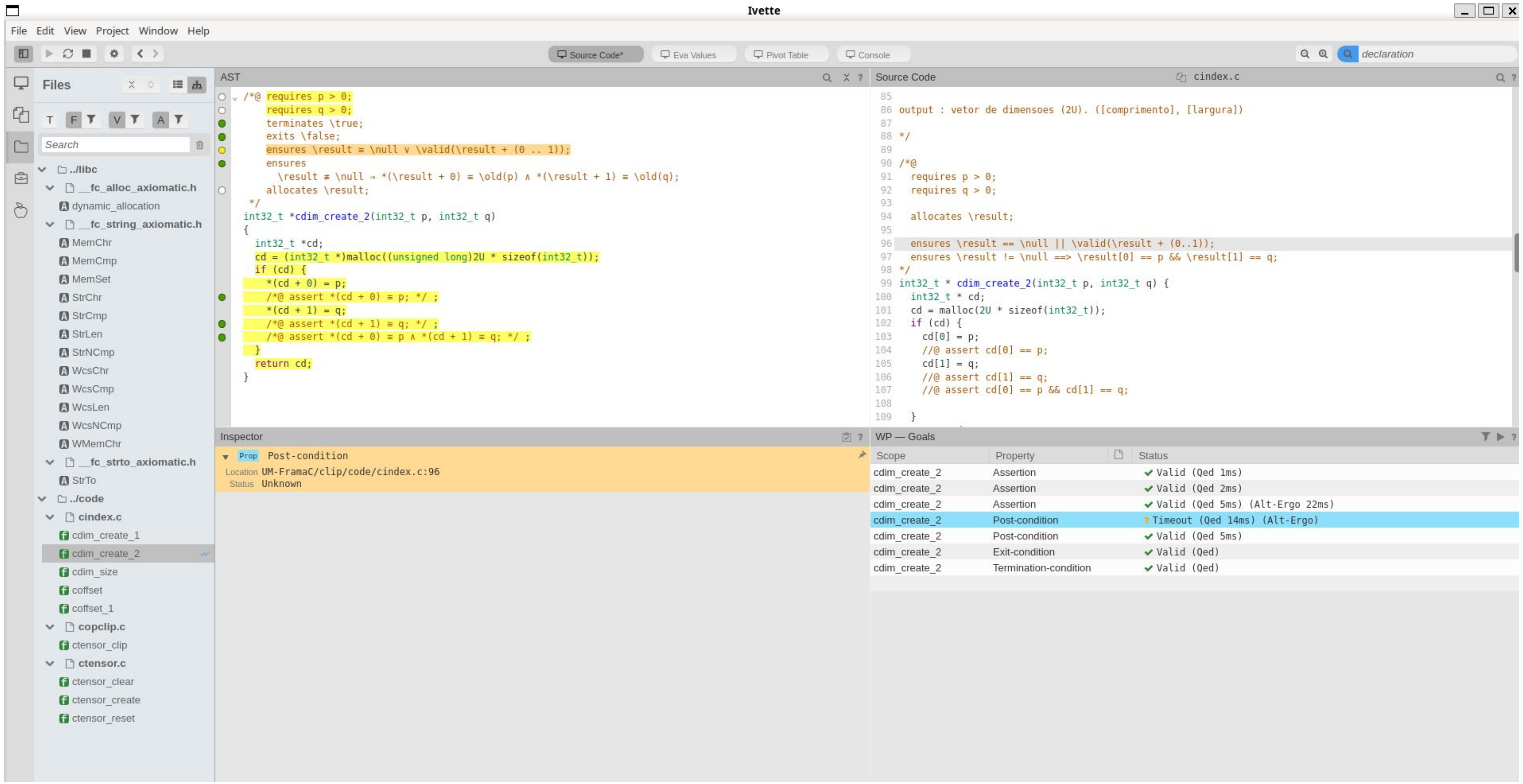
- **Why3 Formal Specification** - What the operator **must** do.
- **C Implementation** (Generated from Why3) - How it actually works (concrete with **pointers and loops**)
- **Frama C Annotations** - The bridge that should respect the regulations stated by Why3's specification and assures **critical safety** in the C implementation.

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 hold throughout execution

Ivette - Interface Gráfica



Command Line

```
gojalo@Caixeiro:~/mei/pmfp/proj/UM-FramaC/clip/code$ frama-c -wp -wp-prover alt-ergo,cvc4,z3 -wp-timeout 30 -wp-fct cten
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)
```

Why3 - C - FramaC

What Why3 states	C Encodes It As	Verification with Frama-C
opwhere is functional	Loop body matches functional definition	Loop invariant + postcondition
Shape compatibility **a~=b**	**rank** equality + dimension array equality	**requires** precondition
Valid tensor	**Abstract Type**	**valid_tensor** predicate (pointer + rank + dims checks)
No side effects	writes only to **r.t_data**	**assigns r.t_data[...]** clause
Separation (implicit in Why3)	Pointers don't overlap	**\separated(...)** precondition
Loop invariant (implict)	Explicit ACSL **loop invariant**	Ghost assert at each iteration

This table represents what each layer must follow, in this case, specified for the **where** operator.

Even though the C code is automatically generated by Why3, the **Frama-C Annotations** need to have the same scope as Why3, meaning:

There shall be no cases forbidden by Why3 allowed by Frama-C

Why3 - C - FramaC

```
function opwhere (cond : tensor bool) (a : tensor real) (b : tensor real)
  : tensor real
  = fun k => if cond.data[k] then a.data[k] else b.data[k]
```

```
int32_t m = cdim_size(r->t_dims, r->t_rank);  // Flat array size
int32_t i, o = m - 1;
if (0 <= o) {
  for (i = 0; ; ++i) {
    r->t_data[i] = (cond->t_data[i] > 0.0)     // Abstract condition
                 ? a->t_data[i]                 // True branch
                 : b->t_data[i];                // False branch
    if (i == o) break;
  }
}
```

Preconditions

```
requires valid_tensor(cond);  // Pointer valid, rank positive, dims positive
requires valid_tensor(a);
requires valid_tensor(b);
requires valid_tensor(r);
requires cond.t_rank == a.t_rank == b.t_rank == r.t_rank;  // Same shape
requires \separated(r.t_data, cond.t_data, a.t_data, b.t_data);  // No aliasing
```

Loop Invariants (captures what's true each iteration)

```
\forall integer k; 0 <= k < i ==>
  r.t_data[k] == ((0.0) < cond.t_data[k] ? a.t_data[k] : b.t_data[k])
```

Postconditions

```
ensures \forall integer i; 0 <= i < dim_size(r) ==>
  r.t_data[i] == ((0.0) < cond.t_data[i] ? a.t_data[i] : b.t_data[i])
```

Code Structure Where

There are 3 modules:

- **cIndex** - Index & Dimension Utilities
  - **cdim_size()** - returns total element count
  - **coffset()** - linear offset from multi-dimensional indices
- **cTensor -** Tensor creation and reset
  - **ctensor_create() -** alloc + init
  - **ctensor_clear() -** wipe Tensor
  - **wraps malloc - handle alloc failure**
- **copWhere -** Where operations
  - **ctensor_where(cond, a, b, r) -** element-wise conditional

$$Z[i] = \begin{cases} X[i] & \text{if } condition[i] = \text{True} \\ Y[i] & \text{otherwise} \end{cases}$$

ACSL Contracts - Where

In every **head** file we need to write the contracts in order to determine :

- ***requires*** **- Precondition (**what must be true when the function is called**)**
- ***assigns*** - What memory the function is allowed to modify
- ***ensures -*** **Postcondition (**guarantees from the output**)**
- **\valid (p + (0..n)) -** Pointer **p[0]** trough **p[n]** are valid (allocated, accessible)

ACSL Axiom - Where

Since ACSL, can't call C functions inside specifications, we defined a **mirror axiom (**axiomatic Dimsize{
}**).**

This axiomatic allowed us to use cdim_size inside specifications.

- **dim_size_logic(int32_t *u, integer n) reads u[0 .. n-1];**
  - takes an array **u** and a count **n, returns an integer**
  - **reads u[0..n-1],** tells the prover the function depends only on the values of u[0] until u[n-1], other values outside that range do not affect the outcome

We cover the following:

- A 0 dimension Tensor size is 1(minimum size)
- The size on n dimensions is the product of the first **n-1** dimensions * n-th dimension.
  - A 3x4x5 tensor has dim_size_logic(u,3) = 3x4x5 = 60
- Total size should never exceed **INT_MAX**, this is needed so WP does not detect unwanted overflows.

ACSL Axiom - Where

```
/*@
axiomatic DimSize {
logic integer dim_size_logic(int32_t *u, integer n) reads u[0 .. n-1];

axiom dim_size_zero:
\forall int32_t *u; dim_size_logic(u, 0) == 1;

axiom dim_size_step:
\forall int32_t *u, integer n; n >= 1 == >
dim_size_logic(u, n) == dim_size_logic(u, n-1) * u[n-1];

axiom dim_size_positive:
\forall int32_t *u, integer n; n >= 0 == > dim_size_logic(u, n)>=1;

axiom dim_size_bounded:
\forall int32_t *u, integer n; n >= 0 == > dim_size_logic(u, n) <=
2147483647;
*/
```

ACSL Verification - Where

The operator receives 4 tensors, for each element i, if cond[i] > 0, write a[i] into r[i], otherwise write b[i] into r[i].

The result tensor (r) is the only entity altered.

- **Rank Checks**
  - A rank cant be negative, and the maximum dimension is 4.
- **The dimensional arrays must be readable**
  - Either the rank is 0 (no arrays, just scalars)
  - Or the dims array must be valid memory pointers from **0 to rank -1**
  - This tells **WP that the memory addresses are valid** (r.t_dims[0] …)**.**
- **Each dimension is positive**
  - for each index **K** in the dimensional array, that dimension must be at least , preventing 0 sized dimensions.
- **Each dimension is bounded**
  - each dimension can be at most 1000, this is an **arbitrary choice**, just to prevent **int32** overflow, which could lead to invalid dim sizes
- **The 4 data Arrays must cover the correct range**
  - **cond, a and b are inputs** therefore they are **not writable.**
  - **dim_size_logic** needs to receive the correct range where memory exists and is safe to use, being **separated** from other tensors.
- **The Frame Condition (output)**
  - States that the only memory that this function touches will be **r.t_data**

ACSL Verification - Where

```
/*@
  // — validity (Why3: valid_tensor cond/a/b/r) ————————————————————
  requires valid_tensor(cond);
  requires valid_tensor(a);
  requires valid_tensor(b);
  requires valid_tensor(r);

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

  // — cond, a, b must be readable over the same flat index range as r ——
  requires \valid_read(cond.t_data + (0 .. dim_size_logic(r.t_dims, r.t_rank) - 1));
  requires \valid_read(a.t_data    + (0 .. dim_size_logic(r.t_dims, r.t_rank) - 1));
  requires \valid_read(b.t_data    + (0 .. dim_size_logic(r.t_dims, r.t_rank) - 1));

  // — no aliasing between r.t_data and any input ————————————————
  // Without these WP must assume writing r.t_data[i] may mutate cond/a/b,
  // which prevents it from proving the loop invariant is preserved and the
  // postcondition holds.
  requires \separated(r.t_data + (0 .. dim_size_logic(r.t_dims, r.t_rank) - 1),
                      cond.t_data + (0 .. dim_size_logic(r.t_dims, r.t_rank) - 1));
  requires \separated(r.t_data + (0 .. dim_size_logic(r.t_dims, r.t_rank) - 1),
                      a.t_data + (0 .. dim_size_logic(r.t_dims, r.t_rank) - 1));
  requires \separated(r.t_data + (0 .. dim_size_logic(r.t_dims, r.t_rank) - 1),
                      b.t_data + (0 .. dim_size_logic(r.t_dims, r.t_rank) - 1));

  // — only r.t_data is modified ————————————————————————
  assigns r.t_data[0 .. dim_size_logic(r.t_dims, r.t_rank) - 1];

  // — functional postcondition (single ITE equality, mirrors loop body) ——
  ensures \forall integer i;
    0 <= i < dim_size_logic(r.t_dims, r.t_rank) ==>
      r.t_data[i] == (((double) 0.0) < cond.t_data[i] ? a.t_data[i] : b.t_data[i]);
*/
void ctensor_where(struct ctensor cond, struct ctensor a, struct ctensor b,
                   struct ctensor r);
```

Testing

In order to test our ACSL specification we ran the following:

```
frama-c -wp -wp-rte -wp-prover alt-ergo -wp-timeout 30 -wp-fct ctensor_where copwhere.c cindex.c
```

verifies the **where operator,** assuming everything else is correct (**cdim_size()** for eg).

To test everything:

```
frama-c -wp -wp-rte -wp-prover alt-ergo -wp-timeout 30
copwhere.c cindex.c
```

All functions in both files were verified, 176 goals were scheduled and 8 Timeouts were detected, all of the timeouts originated from **cdim_create(),** which deals with dynamic memory allocation.

It is documented that the **WP plug-in** can't work with dynamic allocation, meaning that, since **cdim_create()** uses malloc, **WP** can't prove it.

source: https://allan-blanchard.fr/publis/frama-c-wp-tutorial-master-en.pdf

Testing

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:

```c
/*@
  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
```

Code Structure - Clip

There are 3 modules:

- **cIndex** - Index & Dimension Utilities
  - **cdim_size()** - returns total element count
  - **cdim_create_1() / cdim_create_2()**
  - **coffset()**
- **cTensor -** Tensor creation, reset and clear
  - **ctensor_create()**
  - **ctensor_clear()**
  - **ctensor_reset()**
- **copClip - Tool logic**

$$Y = \mathbf{Clip}(X, L, M)$$

where:

- $input$: input tensor (denoted by X)
- min: minimum value (scalar) (denoted by L)
- max: maximum value (scalar) (denoted by M)
- $output$: output tensor (denoted by Y)

Clip - Possible Problems

Possible problems:

- No pointer checking
  - Many of the functions do not check the validity of the vectors used.
  - **cdim_size and coffset**, for example, do not check the vectors that are parameters.
- Parameter validation
  - Negative values
- Overflow

```
int32_t limits: 2147483647 (max) and -2147483648 (min)

--- Test 1: cdim_size with overflow ---
Dimensions: [50000, 50000]
Expected: 2,500,000,000 (greater than INT32_MAX)
Result: -1794967296
Overflow? YES
```

Clip - ctensor Verification

- **ctensor_create:**
  - **Negative n.**
  - **valid_read** - will prevent segmentation faults when accessing the vector.
  - **Valid dimensions** - the requires function ensures that the dimensions present in the tensor are positive.
  - **allocates** - tells what will be allocated
  - **Pre/Post-conditions**
- **ctensor_clear & ctensor_reset:**
  - **assigns** - r.data will be changed
  - **loop invariants**
  - **loop assigns**
  - **loop variant**

Clip - ctensor Verification

```
/*@
  requires n > 0;
  requires \valid_read(ds + (0..n-1));
  requires \forall integer j; 0 <= j < n ==> ds[j] >= 1;

  allocates \result.t_data;

  ensures \result.t_dims == ds;
  ensures (\result.t_rank == n && \result.t_data != \null) ||
          (\result.t_rank == 0 && \result.t_data == \null);
*/

struct ctensor ctensor_create(int32_t * ds, int32_t n) {
  int32_t m;
  double * vs;
  struct ctensor ctensor;
  m = cdim_size(ds, n);
  vs = malloc(((uint32_t) m) * sizeof(double));
  ctensor.t_rank = !vs ? 0 : n;
  ctensor.t_dims = ds;
  ctensor.t_data = vs;
  return ctensor;
}
```

```
/*@
  requires r.t_rank > 0;
  requires \valid_read(r.t_dims + (0..r.t_rank-1));
  requires \forall integer j; 0 <= j < r.t_rank ==> r.t_dims[j] >= 1;
  requires r.t_data != \null;
  assigns r.t_data[..];
*/
void ctensor_reset(struct ctensor r, double v) {
  int32_t m, i, o;
  m = cdim_size(r.t_dims, r.t_rank);
  //@ assert m >= 1;
  o = m - 1;
  //@ assert o >= 0;
  if (0 <= o) {
    /*@
      loop invariant 0 <= i <= o;
      loop invariant \forall integer k; 0 <= k < i ==> r.t_data[k] == v;
      loop assigns i, r.t_data[0..o];
      loop variant o - i;
    */
    for (i = 0; ; ++i) {
      r.t_data[i] = v;
      if (i == o) {
        break;
      }
```

Clip - cindex Verification

- **cdim_size:**
  - **Negative n.**
  - **valid_read** - will prevent segmentation faults when accessing the vector.
  - **Valid dimensions -** the requires function ensures that the dimensions present in the tensor are positive.
  - **allocates** - tells what will be allocated
  - **Post-conditions**
- **coffset:**
  - **valid_read** on ks and ds
  - **loop invariants**
  - **loop assigns**
  - **loop variant**

Clip - cindex Verification

```
/*@
  requires n > 0;
  requires \valid_read(u + (0..n-1));
  requires \forall integer j; 0 <= j < n ==> u[j] >= 1;
  ensures \result >= 1 ;
  assigns \nothing;
*/
int32_t cdim_size(int32_t * u, int32_t n) {
  int32_t p;
  int32_t i, o;
  p = 1;
  o = n - 1;
  if (0 <= o) {

    /*@
    loop invariant 0 <= i <= o;
    loop invariant p >=1;
    loop assigns i, p;
    loop variant o - i +1 ;
    */
    for (i = 0; ; ++i) {
      p = p * u[i];
      if (i == o) {
        break;
      }
    }
  }
  return p;
}
```

```
/*@
  requires n > 0;
  allocates \result;

  ensures \result == \null || \valid(\result + (0..0));
  ensures \result != \null ==> \result[0] == n;
*/

int32_t * cdim_create_1(int32_t n) {
  int32_t * cd;
  cd = malloc(1U * sizeof(int32_t));
  if (cd) {
    //@ assert \valid(cd + (0..0));
    cd[0] = n;
    //@ assert cd[0] == n;
  }
  //@ assert cd == \null || (cd != \null && cd[0] == n);
  return cd;
}
```

```
/*@
  requires n >= 0;
  requires n > 0 ==> \valid_read(ks + (0..n-1));
  requires n > 0 ==> \valid_read(ds + (0..n-1));
  requires \forall integer j; 0 <= j < n ==> ds[j] > 0;

  assigns \nothing;
  ensures \result >= -1;
*/
int32_t coffset(int32_t * ks, int32_t * ds, int32_t n) {
  int32_t p;
  int32_t i, o, d, k;
  p = 0;
  o = n - 1;
  if (0 <= o) {
    /*@
      loop invariant 0 <= i <= o;
      loop invariant p >= 0;
      loop assigns i, p, d, k;
      loop variant o - i + 1;
    */
    for (i = 0; ; ++i) {
      d = ds[i];
      k = ks[i];
      if (0 <= k && k < d) {
        p = p * d + k;
      } else {
        return -1;
      }
      if (i == o) {
        break;
      }
    }
  }
  return p;
}
```

Clip - copclip Verification

- **ctensor_clip:**
  - **Negative values**
  - **valid_read**
  - **Valid dimensions**
  - **Limits verification**
  - **Pre/Post-conditions**
  - **Null Pointers**
  - **Loop invariants**
  - **Loop assigns**
  - **Loop variants**

Clip - copclip Verification

```
/*@
  requires x.t_rank > 0;
  requires \valid_read(x.t_dims + (0..x.t_rank-1));
  requires \forall integer j; 0 <= j < x.t_rank ==> x.t_dims[j] >= 1;
  requires x.t_data != \null;

  requires l.t_rank > 0;
  requires l.t_data != \null;

  requires m.t_rank > 0;
  requires m.t_data != \null;

  requires r.t_rank > 0;
  requires \valid_read(r.t_dims + (0..r.t_rank-1));
  requires \forall integer j; 0 <= j < r.t_rank ==> r.t_dims[j] >= 1;
  requires r.t_data != \null;

  requires x.t_rank == r.t_rank;
  requires \forall integer j; 0 <= j < r.t_rank ==> x.t_dims[j] == r.t_dims[j];

  assigns r.t_data[..];
*/
```

```
void ctensor_clip(struct ctensor x, struct ctensor l, struct ctensor m,
                  struct ctensor r) {
  int32_t n, i, o;
  double l_background, m_background;

  n = cdim_size(r.t_dims, r.t_rank);

  //@ assert n >= 1;

  l_background = l.t_data[0];
  m_background = m.t_data[0];

  o = n - 1;
  //@ assert o >= 0;

  if (0 <= o) {
    /*@
      loop invariant 0 <= i <= o;
      loop assigns i, r.t_data[0..o];
      loop variant o - i;
    */
    for (i = 0; ; ++i) {
      r.t_data[i] = (fmin(m_background, (fmax((x.t_data[i]), l_background))));
      if (i == o) {
        break;
```

Testing

In order to test our ACSL specification we ran the following:

```
frama-c -wp -wp-prover alt-ergo,cvc4,z3 -wp-timeout 30 -wp-fct ctensor_clear ctensor.c cindex.c
```

Provers :

- alt-ergo
- cvc4
- z3

Timeout increase:

10s → 30s

cdim_size: 15/15 goals proved

coffset: 19/19 goals proved

ctensor_create: 7/7 goals proved

ctensor_clear: 23/23 goals proved

ctensor_reset: 23/23 goals proved

ctensor_clip: 32/32 goals proved

Timeouts caused by memory allocation:

cdim_create_1: 6/7 goals proved (1 timeout)

cdim_create_2: 6/7 goals proved (1 timeout)

Conclusions

- Successfully used Frama-C WP to verify the Where and Clip operators' safety properties
- Every contract was demonstrated to have no overflow, out-of-bounds, or null pointers.
- Limitations: WP's documented inability to validate dynamic allocation (malloc)

For the Future

- Add functional correctness verification to current operators to demonstrate that the output values are mathematically valid rather than merely memory-safe.
- Extend verification to other ONNX operators (Add, MatMul, etc.).