

ClearSy Hackathon 2024

João Pedro de A. Paula

July 12, 2024

Contents

Day 1: Abstract Iterator	1
Day 2: Specification of block occupancy	4
Necessary occupancy property	4
unmask_blocks operation	4
set_tdl_alarm operation	5
mask_blocks operation	6
release_blocks operation	7
Day 3: Implementation of block occupancy operations	7
unmask_blocks loop invariant	8
set_tdl_alarm implementation	8
mask_blocks implementations	9
release_blocks implementation	10
Day 4: Implementation of services and iterator	11
block_miniservices_i.imp's initialisation and implementation . .	11
read_ob, read_tdl_a and read_mb	12
(un)occ_blocks, (un)mask_blocks and (un)alarm_blocks .	12
Rest of the query operations	13
block_occupancy_it_i.imp's initialisation and implementation . .	17
Proof obligations and B0 check screenshots	ATTACH 18

Day 1: Abstract Iterator

We were tasked with providing the response (line 13), as well as the loop's linking invariant (line 19).

```

1  res <-- all_MP85 =
2  VAR
3    current, continue, current_is_MP85
4  IN
5    current_is_MP85 := FALSE;
6    res := TRUE;
7    continue <-- init_iter;
8    WHILE
9      continue = TRUE & res = TRUE
10   DO
11     continue, current <-- next_iter;
12     current_is_MP85 <-- is_MP85_op(current);
13     IF current_is_MP85 = FALSE
14     THEN res := FALSE
15     END
16   INVARIANT
17     Todo \ / Done = Trains &
18     continue = bool (Todo /= {}) &
19     res = bool((Done <| is_MP85)~[{TRUE}] = Done)
20   VARIANT
21     card(Todo)
22   END
23 END

```

The response is pretty straight-forward, we just need to check whether the current train is an MP85 and, when it isn't, set the variable `res` to the value `FALSE`; when `res` is set to `FALSE` it will never be changed to `TRUE`, and if all of the trains are MP85s then the `IF` condition will never be met and `res` will remain `TRUE`, giving us the correct result.

The linking invariant is a bit more complex. We need to check that all of the trains we already iterated over (`Done <| is_MP85`) are mapped to `TRUE`. The original invariant asserted over all of the trains, but we need to restrict ourselves to only the trains that we already checked.

To prove our operation is indeed correct, the main thing is to prove our loop terminates with the correct response.

At the start of the loop that is easy to see, since `Done` is started as the empty set, and `Todo = Trains`, all of the invariants are clearly respect. The first two clauses of the invariant will be assumed to be true, since they were already proven in their respective machines, and we aren't altering state beyond calling the operations. Let us focus on the part that was fulfilled as

the assignment for this hackathon i.e. we need to assert that, before entering the loop,

$$\mathbf{res} = \mathbf{TRUE} \iff ((\mathbf{Done} \triangleleft \mathbf{is_MP85})^{-1}[\{\mathbf{TRUE}\}] = \mathbf{Done})$$

and since \mathbf{Done} starts as \emptyset , $\mathbf{res} := \mathbf{TRUE}$ and $(\emptyset \triangleleft \mathbf{is_MP85})^{-1}[\{\mathbf{TRUE}\}] = \emptyset$ —because $\emptyset \triangleleft R = \emptyset$ for any relation R —, the invariant is respected before the start of the loop.

Moreover, at each execution we can verify that \mathbf{res} has the correct value as well. We have a couple of cases to verify:

Case $\mathbf{current_is_mp85} = \mathbf{TRUE}$: In this case, since we already checked the invariant for $\mathbf{Done} \cup \{\mathbf{current}\}$, $\mathbf{Done} = \mathbf{Done} \cup \{\mathbf{current}\}$, and we know that $\mathbf{is_MP85}(\mathbf{current}) = \mathbf{TRUE}$, indeed

$$(((\mathbf{Done} \cup \{\mathbf{current}\}) \triangleleft \mathbf{is_MP85})^{-1}[\{\mathbf{TRUE}\}] = \mathbf{Done} \cup \{\mathbf{current}\})$$

Thus, given that $\mathbf{res} = \mathbf{TRUE}$, the invariant is respected.

Case $\mathbf{current_is_MP85} = \mathbf{FALSE}$: Here, we have that $\mathbf{Done} = \mathbf{Done} \cup \{\mathbf{current}\}$ and $\mathbf{is_MP85}(\mathbf{current}) = \mathbf{FALSE}$, thus

$$(\mathbf{Done} \triangleleft \mathbf{is_MP85})^{-1}[\{\mathbf{TRUE}\}] = \mathbf{Done} - \{\mathbf{current}\} \neq \mathbf{Done}$$

i.e. it is false. Therefore the invariant is still respected, since $\mathbf{res} := \mathbf{FALSE}$.

There is also the obligation of showing that the invariant remains respected at the end of the loop execution. It has two stop conditions:

Either $\mathbf{res} = \mathbf{FALSE}$: In which case the argument is the same as when $\mathbf{current_is_MP85} = \mathbf{FALSE}$.

Or $\mathbf{continue} = \mathbf{FALSE}$: This case means that we've exhausted the whole set and all trains were asserted to be MP85s (otherwise the loop would've stopped given the previous stop condition), at which point $\mathbf{Done} = \mathbf{Trains}$ and $\mathbf{is_MP85}[\mathbf{Trains}] = \{\mathbf{TRUE}\}$. Therefore

$$(\mathbf{Trains} \triangleleft \mathbf{is_MP85})^{-1}[\{\mathbf{TRUE}\}] = \mathbf{is_MP85}^{-1}[\{\mathbf{TRUE}\}] = \mathbf{Trains}$$

which is what we wanted, given that \mathbf{res} remains \mathbf{TRUE} .

Day 2: Specification of block occupancy

For the second day of the hackathon, we were tasked with completing the abstract specification of the occupancy of a block. That entails specifying a couple of operations and a property that states a necessary condition for the occupancy of a block.

Necessary occupancy property

The natural language specification states

A block having one of its border detector occupied or having its trackside detector occupied has to be occupied.

and we need to complete that property in the entry-point operations `set_block_occupancy` and `execute_cycle`. The B specification is

```
d_b2b[obd] \ / otd <: ob
```

where a `d_b2b[obd]` gives us the blocks where one of its border detectors is occupied and `otd` is the set of occupied trackside detectors, so any block that is in either of these sets should also be in the set of occupied blocks.

unmask_blocks operation

This operations should remove from the set of masked blocks any block that is free or when all of the following conditions are true:

1. The upward block has a free trackside detector or the upward block is free.
2. The downward block has a free trackside detector or the downward block is free.

So, we must leave all previously masked blocks that don't match these conditions alone, and remove all blocks matching 1) and 2):

```
1 unmask_blocks =
2 BEGIN
3   // leave all previously masked blocks, but unmask
4   mb := mb - (
5     // blocks that are free (d_free_b), or
```

```

6      d_free_b \/  

7      // 1) whose upward block (cfg_b2b_up~) has a free trackside detector or  

8      // is free (d_free_td \/  

9      (cfg_b2b_up~[d_free_td \/  

10     // 2) whose downward block (cfg_b2b_down~) has a free trackside detector  

11     // or is free (d_free_td \/  

12     cfg_b2b_down~[d_free_td \/  

13 )  

14 END

```

We are using `mb` as a starting point in order to leave masked blocks that shouldn't be unmasked alone. In line 6 we are removing any free block, or blocks where **both** (represented by the intersection) either the upward trackside detector is free (`d_free_tb`) or the block itself is free (`d_free_b`) (line 9, note how we're using the inverse of the relation from blocks to upward blocks, i.e. given an upward block, get the block itself), and the same for the downward block (line 12).

set_tdl_alarm operation

As the name implies, `set_tdl_alarm` is responsible for set the Trackside Detector Loss alarm to all blocks that

1. Are occupied.
2. Are not masked.
3. Have its trackside detector free.

That is, all blocks that should've been occupied given the other information, but whose trackside detector is still free. This operation shouldn't remove any block from the alarm, only add blocks that match the conditions.

```

set_tdl_alarm =  

BEGIN  

  // tdl remains unchanged, but set the alarm to all blocks that  

  tdl := tdl \/  

  // 1) are occupied, 3) have free trackside detectors (ob /\ d_free_td)  

  // and 2) are not masked (- mb)  

  (ob /\ d_free_td) - mb  

  // Remember to prove that (ob /\ (t_block - otd)) - mb = ob - mb - otd  

  //
  d_free_td

```

```
)
END
```

Explaining it, $\mathbf{ob} \cap \mathbf{d_free_td}$ is the set of all occupied blocks with free trackside detectors. The result of that minus \mathbf{mb} is precisely what was specified, blocks that are occupied with free trackside detectors and aren't masked.

The condition specified in `block_services.mch` is $(\mathbf{ob} \setminus \mathbf{mb}) \setminus \mathbf{otd}$, which we shall prove is equivalent to the specification above:

$$\begin{aligned}
x \in (\mathbf{ob} \cap (\mathbf{t_block} \setminus \mathbf{otd})) \setminus \mathbf{mb} &\Leftrightarrow x \in \mathbf{ob} \wedge x \in \mathbf{t_block} \wedge x \notin \mathbf{otd} \wedge x \notin \mathbf{mb} \\
&\Leftrightarrow x \in \mathbf{ob} \wedge x \notin \mathbf{otd} \wedge x \notin \mathbf{mb} \\
&\Leftrightarrow x \in \mathbf{ob} \wedge x \notin \mathbf{mb} \wedge x \notin \mathbf{otd} \\
&\Leftrightarrow x \in (\mathbf{ob} \setminus \mathbf{mb}) \wedge x \notin \mathbf{otd} \\
&\Leftrightarrow x \in (\mathbf{ob} \setminus \mathbf{mb}) \setminus \mathbf{otd}
\end{aligned}$$

mask_blocks operation

The `mask_blocks` operation is responsible for masking any blocks that aren't alarming and whose one of its borders is occupied, in order to not trigger an alarm when a train is leaving or entering the block. In other words, if a block is masked we don't do anything, but if it

1. Is not in TDL alarm; and
2. One of the block borders is occupied.

it should be included in \mathbf{mb} .

```
mask_blocks =
BEGIN
  // leave mb unchanged, but mask all blocks where
  mb := mb \ / (
    // 2) one of its borders (d_bd2b) is occupied (obd) and 1) is not in TDL
    // alarm (- tdla)
    d_bd2b[obd] - tdla
  )
END
```

The specification is grabbing all blocks with occupied border detectors (`d_bd2b[obd]`), and removing blocks that have an alarm triggered (`tdla`).

release_blocks operation

Lastly, but not least, `release_blocks` should remove from the set of occupied blocks any block meeting **all** of the following conditions:

1. The block is not in TDL alarm or the block is being initialized by the Control Center.
2. A block exit detector, which was occupied during the previous cycle, is now released.

Which is represented in B as:

```
release_blocks =
BEGIN
  // leave occupied blocks as is, but release blocks that
  ob := ob - (
    // 1) are not in TDL alarm or being initialized ((t_block - tdla) \ cc_init), and
    ((t_block - tdla) \ cc_init) /\
    // 2) had an exit previously occupied, but not anymore (oed_prev - oed,
    // d_be2b is used to grab the blocks for those exit detectors)
    d_be2b[oed_prev - oed]
  )
END
```

So $(t_block \setminus tdla) \cup cc_init$ is the set of blocks that are not alarming plus the ones being initialized by the Control Center; and $oed_prev \setminus oed$ represents all previously occupied exit detectors that are now free, so $d_be2b[oed_prev \setminus oed]$ should give us the blocks matching those exit detectors. Intersecting that with the previous set gives us all blocks matching the specified conditions.

Day 3: Implementation of block occupancy operations

For the third day we were tasked with implementing the operations specified in `block_occupancy.mch`.

Most of the loop invariant (if not all), have left some unproven POs regarding whether the addition of a new block to the set of treated blocks. In fact, all of them could be proved but for the sake of time limitation I'll try to provide informal arguments as to why the invariants are indeed correct.

Since the invariants are roughly a copy of the abstract specification, which is proven to be correct, but restricted, via the intersection, to the set of treated blocks, we can be sure that the resulting set at each iteration will be precisely the treated blocks that match the specification's conditions.

unmask_blocks loop invariant

This operation's loop implementation was given as an example, all we had to do was to complete its invariant, which looks like

```
// At each iteration we're removing the treated blocks that satisfy the
// condition to unmask from the set of masked blocks
mb = mb$0 - (
  // Get all blocks that satisfy the condition for unmasking
  (d_free_b \
    (cfg_b2b_up~[d_free_td \ d_free_b] /\ cfg_b2b_down~[d_free_td \ d_free_b]))
  // and only consider those that were already treated
  /\ treated_block
)
```

Here, `mb$0` refers to the value of `mb` in the last iteration of the loop. So `mb` should be precisely the blocks that were masked previously minus the matching the abstract specification, the only catch is that we need to filter out, via the intersection, for the blocks that were already treated, since we can't assert anything for the ones that we haven't iterated over.

set_tdl_alarm implementation

For the rest of the operations we had to provide the full loop implementation. Most of it is straight-forward, we are just grabbing the next value via the iteration operations, verifying whether we still have blocks to treat and using some service operations to check for the abstract conditions.

```
set_tdl_alarm = VAR cont IN
  cont <-- init_iteration_t_block;
  WHILE cont = TRUE DO
    VAR block, is_alarming, should_alarm IN
      cont, block <-- iterate_t_block;
      is_alarming <-- read_tdl(block);
      // If the block is already alarming, don't do anything
      IF is_alarming = FALSE THEN
```



```

        should_alarm <-- cond_alarm(block);
        // Otherwise check if the alarming condition was met
        IF should_alarm = TRUE THEN
            alarm_block(block)
        END
    END
END
INVARIANT
    cont = bool(block_to_treat /= {}) &
    block_to_treat <: t_block &
    treated_block <: t_block &
    block_to_treat /\ treated_block = {} &
    block_to_treat \/ treated_block = t_block &
    tdla <: t_block &
    // All of the treated blocks that match the alarming condition should be
    // included in the alarm set
    tdla = tdla$0 \/ (((ob /\ d_free_td) - mb) /\ treated_block)
VARIANT
    card(block_to_treat)
END
END

```

Once again the loop invariant just asserts some iterator properties plus the fact that the last iteration should have only added to `tdla` the treated blocks that match the conditions.

mask_blocks implementations

Similarly, the implementation of `mask_blocks` will differ from the other only with regards to the sets and conditions being checked.

```

mask_blocks = VAR cont IN
    cont <-- init_iteration_t_block;
    WHILE cont = TRUE DO
        VAR block, is_masked, should_mask IN
            cont, block <-- iterate_t_block;
            is_masked <-- read_mb(block);
            // If the block is already masked, don't do anything
            IF is_masked = FALSE THEN
                should_mask <-- cond_mask(block);
            END
        END
    END
END

```

```

        // Otherwise check if the masking condition was met
        IF should_mask = TRUE THEN
            mask_block(block)
        END
    END
END
INARIANT
    cont = bool(block_to_treat /= {}) &
    block_to_treat <: t_block &
    treated_block <: t_block &
    block_to_treat /\ treated_block = {} &
    block_to_treat \/ treated_block = t_block &
    mb <: t_block &
    // All of the treated blocks that match the masking condition should be
    // included in the masked blocks set
    mb = mb$0 \/ ((d_bd2b[obd] - tdl_a) /\ treated_block)
VARIANT
    card(block_to_treat)
END
END

```

And the invariant asserts that `mb` is precisely the set of all treated blocks where one of its border detectors is occupied and that are not in TDL alarm.

release_blocks implementation

Finally, here's the implementation of `release_blocks` (I won't expand much on it, the explanation is very similar to the previous ones):

```

release_blocks = VAR cont IN
    cont <-- init_iteration_t_block;
    WHILE cont = TRUE DO
        VAR block, is_released, should_release IN
            cont, block <-- iterate_t_block;
            is_released <-- cond_release(block);
            // If the block is already released, don't do anything
            IF is_released = FALSE THEN
                should_release <-- cond_release(block);
                // Otherwise, check if the release condition was met
                IF should_release = TRUE THEN

```

```

        unocc_block(block)
    END
END
END
INVARIANT
    cont = bool(block_to_treat /= {}) &
    block_to_treat <: t_block &
    treated_block <: t_block &
    block_to_treat /\ treated_block = {} &
    block_to_treat \/ treated_block = t_block &
    ob <: t_block &
    ob = ob$0 - (
        // All treated blocks that match the release condition should not be
        // included in the occupied blocks set
        (((t_block - tdla) \/ cc_init) /\ d_be2b[oed_prev - oed])
        /\ treated_block
    )
VARIANT
    card(block_to_treat)
END
END

```

Day 4: Implementation of services and iterator

The fourth day was the last one with new challenges. This time we had to implement all of the helper operations used in the block occupancy implementation, alongside with the iterator's implementation.

`block_miniservices_i.imp`'s initialisation and implementation

Given that we're implementing the sets of blocks as tables—where if the element is present in the abstract set, its index in the implementation table will be set to true—, then we need to initialize the implementation tables like this:

```

tab_ob := t_block_i * {TRUE};
tab_tdla := t_block_i * {TRUE};
tab_mb := t_block_i * {FALSE}

```

This means that we are initializing the occupied blocks and the TDL alarm tables such that every block is considered to be occupied and alarming;

on the other hand, every block is initialized as unmasked. We can show that this initialisation respects the invariant, since

$$\text{tab_ob}^{-1}[\{\text{TRUE}\}] \cap \mathbf{t_block} = \mathbf{t_block_i} \cap \mathbf{t_block} = \mathbf{t_block} = \text{ob}$$

The same argument holds for `tab_tdla`. In the case of `tab_mb`,

$$\text{tab_mb}^{-1}[\{\text{TRUE}\}] \cap \mathbf{t_block} = \emptyset \cap \mathbf{t_block} = \emptyset = \text{mb}$$

`read_ob`, `read_tdla` and `read_mb`

Implementing those query operations can be done with a simple one-liner, given the way the sets were implemented:

```
p_res <-- read_ob(p_block) = p_res := tab_ob(p_block);
p_res <-- read_tdla(p_block) = p_res := tab_tdla(p_block);
p_res <-- read_mb(p_block) = p_res := tab_mb(p_block);
```

And in order to prove that this implementation corresponds to the specification, without loss of generality, consider X to be any of `ob`, `tdla` or `mb`. The proof obligation that remains is that $\text{tab_}X(\mathbf{p_block}) = \text{bool}(\mathbf{p_block} \in X)$ and that can be verified with

$$\begin{aligned} \text{tab_}X(\mathbf{p_block}) = \text{TRUE} &\Leftrightarrow \mathbf{p_block} \in \text{tab_}X[\{\text{TRUE}\}] \\ &\Leftrightarrow \mathbf{p_block} \in \text{tab_}X[\{\text{TRUE}\}] \cap \mathbf{t_block} \quad [\mathbf{p_block} \in \mathbf{t_block}] \\ &\Leftrightarrow \mathbf{p_block} \in X \quad [\text{def. of } X] \end{aligned}$$

`(un)occ_blocks`, `(un)mask_blocks` and `(un)alarm_blocks`

Similar to the query operations, these are simple one-liners as well, where we just need to set the value of `p_block` to either `TRUE` or `FALSE`.

```
unocc_block(p_block) = tab_ob(p_block) := FALSE;
occ_block(p_block) = tab_ob(p_block) := TRUE;
unalarm_block(p_block) = tab_tdla(p_block) := FALSE;
alarm_block(p_block) = tab_tdla(p_block) := TRUE;
unmask_block(p_block) = tab_mb(p_block) := FALSE;
mask_block(p_block) = tab_mb(p_block) := TRUE;
```

Some proof obligations still remain. In a similar fashion to the previous section, consider X to be any of `ob`, `tdla` or `mb`. For the operations that remove elements from X we must verify that

$$\begin{aligned} X \setminus \{p_block\} &= (\text{tab_}X^{-1}[\{\text{TRUE}\}] \cap \text{t_block}) \setminus \{p_block\} \\ &= (\text{tab_}X^{-1}[\{\text{TRUE}\}] \setminus \{p_block\}) \cap \text{t_block} \\ &= (\text{tab_}X \triangleleft \{p_block \mapsto \text{FALSE}\})^{-1}[\{\text{TRUE}\}] \cap \text{t_block} \end{aligned}$$

As for operations that add elements to X , we have to check

$$\begin{aligned} (\text{tab_}X \triangleleft \{p_block \mapsto \text{TRUE}\})^{-1}[\{\text{TRUE}\}] \cap \text{t_block} &= (\text{tab_}X^{-1}[\{\text{TRUE}\}] \cup \{p_block\}) \cap \text{t_block} \\ &= (X \cup \{p_block\}) \cap \text{t_block} \\ &= X \cup \{p_block\} \end{aligned}$$

Rest of the query operations

These operations are not so straightforward to implement, but all of them leave no unproven POs. They're implementing parts of the conditions for occupying, masking or alarming blocks, and they all respond with either `TRUE` or `FALSE`. For that reason, I chose to always initialize the value of `p_res` to `FALSE` because, according to the specification, it should **only** be `TRUE` if the conditions are satisfied, otherwise `p_block` isn't a member of the specification's sets we're trying to implement the behaviour for.

1. `is_free_block`

The first query checks whether a block is free, and for that all we need to do is to check whether `tab_ob(p_block) = FALSE`, which would mean the block is *not* occupied.

```
p_res <-- is_free_block(p_block) =
VAR is_occupied IN
  p_res := FALSE;
  is_occupied := tab_ob(p_block);

  // tab_ob(p_block) = FALSE means p_block is *not* occupied
  IF is_occupied = FALSE THEN
    p_res := TRUE
  END
END;
```

2. has_up_free_or_freetd and has_down_free_or_freetd

These two queries check whether there is a free block or trackside detector upward (downward) from the current `p_block`. We start by checking whether `p_block` has any block upwards (downward) from it; if there is, we grab its trackside detector and consult whether it is occupied. In case either the upward (downward) block or its trackside detector are occupied, we respond with `TRUE`; on the contrary, the response will be its default value of `FALSE`.

```
p_res <-- has_up_free_or_freetd(p_block) =
VAR has_up_block, up_block IN
  p_res := FALSE;
  has_up_block, up_block <-- read_all_cfg_b2b_up(p_block);

  IF has_up_block = TRUE THEN
    VAR is_up_free, is_up_td_occupied IN
      is_up_free := tab_ob(up_block);
      is_up_td_occupied <-- read_otd(up_block);

      IF is_up_free = TRUE or is_up_td_occupied = FALSE THEN
        p_res := TRUE
      END
    END
  END
END;

p_res <-- has_down_free_or_freetd(p_block) =
VAR has_down_block, down_block IN
  p_res := FALSE;
  has_down_block, down_block <-- read_all_cfg_b2b_down(p_block);

  IF has_down_block = TRUE THEN
    VAR is_down_free, is_down_td_occupied IN
      is_down_free := tab_ob(down_block);
      is_down_td_occupied <-- read_otd(down_block);

      IF is_down_free = TRUE or is_down_td_occupied = FALSE THEN
        p_res := TRUE
      ELSE
```

```

        p_res := FALSE
    END
END
END
END;

```

3. has_occupied_bd

```

p_res <-- has_occupied_bd(p_block) =
VAR has_up_bd, up_bd, has_down_bd, down_bd, is_up_bd_occupied, is_down_bd_occupied
    p_res := FALSE;
    // We start by getting all upward and downward border detectors,
    has_up_bd, up_bd <-- read_all_cfg_b2bd_up(p_block);
    has_down_bd, down_bd <-- read_all_cfg_b2bd_down(p_block);
    // and initializing them to a default value of FALSE. If the block doesn't
    // have an upward (or downward) border detector, then it won't be a member
    // of the set obd, so read_obd() will reply with FALSE
    is_up_bd_occupied := FALSE;
    is_down_bd_occupied := FALSE;

    // If an upward border detector exists, check whether it is occupied
    IF has_up_bd = TRUE THEN
        is_up_bd_occupied <-- read_obd(up_bd)
    END;

    // Same for downward border detectors
    IF has_down_bd = TRUE THEN
        is_down_bd_occupied <-- read_obd(down_bd)
    END;

    // And if any of them are indeed occupied, the response should be TRUE
    IF is_up_bd_occupied = TRUE or is_down_bd_occupied = TRUE THEN
        p_res := TRUE
    END
END;

```

4. is_occ_unmasked_block

```

p_res <-- is_occ_unmasked_block(p_block) =

```

```

VAR is_occupied, is_masked IN
  p_res := FALSE;
  is_occupied := tab_ob(p_block);
  is_masked := tab_mb(p_block);

  IF is_occupied = TRUE & is_masked = FALSE THEN
    p_res := TRUE
  END
END;

```

5. is_init_or_unalarmed_block

```

p_res <-- is_init_or_unalarmed_block(p_block) =
VAR is_init, is_alarmed IN
  p_res := FALSE;
  is_init <-- read_cc_init(p_block);
  is_alarmed := tab_tdla(p_block);

  IF is_init = TRUE or is_alarmed = FALSE THEN
    p_res := TRUE
  END
END;

```

6. has_up_tr_ed and has_down_tr_ed

```

p_res <-- has_up_tr_ed(p_block) =
VAR has_up_exit, up_exit IN
  p_res := FALSE;
  has_up_exit, up_exit <-- read_all_cfg_b2ed_up(p_block);

  // We start by checking whether there is any upwards exit detector
  IF has_up_exit = TRUE THEN
    VAR is_newly_released IN
      is_newly_released <-- is_exit_newly_released(up_exit);

      // And if there is, according to the specification, we check whether it
      // was released in the previous cycle
      IF is_newly_released = TRUE THEN
        p_res := TRUE
      END
    END
  END
END;

```



```

        END
    END
END
END;

// Same explanation as the operation above, but on the downward direction
p_res <-- has_down_tr_ed(p_block) =
VAR has_down_exit, down_exit IN
p_res := FALSE;
    has_down_exit, down_exit <-- read_all_cfg_b2ed_down(p_block);

    IF has_down_exit = TRUE THEN
        VAR is_newly_released IN
            is_newly_released <-- is_exit_newly_released(down_exit);

            IF is_newly_released = TRUE THEN
                p_res := TRUE
            END
        END
    END
END
END

```

block_occupancy_it_i.imp's initialisation and implementation

Given the invariant

```

rg_t_block_current : (0..size(bijection_t_block)) &
block_to_treat = bijection_t_block[(1..rg_t_block_current)]

```

we'll be iterating over the set from `card(t_block)` to 1, which means that the initialisation of `rg_t_block_current` is

```

rg_t_block_current <-- read_card_t_block

```

The implementation of the operations is pretty straightforward... When starting the iteration, we need to reset `rg_t_block_current` to `card(t_block)` and the condition for continuing the iteration is `rg_t_block_current ≥ 1`, because when it reaches 0,

```

block_to_treat = bijection_t_block[(1..0)] = bijection_t_block[∅] = ∅

```

```

cond <-- init_iteration_t_block =
BEGIN
  rg_t_block_current <-- read_card_t_block;
  cond := bool(rg_t_block_current >= 1)
END;

```

This leaves some POs unproven:

$\text{card}(\text{t_block}) \in 0..\text{size}(\text{bijection_t_block})$ Which is easy to see since $\text{card}(\text{t_block})$ is either 0 or $\text{size}(\text{bijection_t_block})$.

$\text{t_block} = \text{bijection_t_block}[1..\text{card}(\text{t_block})]$ Given that $\text{bijection_t_block} = \text{perm}(\text{t_block})$, $\text{bijection_t_block}[1..\text{card}(\text{t_block})] = \text{ran}(\text{bijection_t_block}) = \text{t_block}$.

And the other operation, the one that actually does the iteration, is implemented as

```






cond, bl <-- iterate_t_block =
BEGIN
  bl <-- read_bijection_t_block(rg_t_block_current);
  rg_t_block_current := rg_t_block_current - 1;
  cond := bool(rg_t_block_current >= 1)
END

```

That is, we get the block for the current index, decrease the index and check the condition again.

These implementations have also left some unproven POs which I'll proof-by-hand-waving here. If the set of blocks to treat is not empty, we need to ensure that $\text{rg_t_block_current}$ is part of the domain of bijection_t_block , which is easily demonstrated by the definition of each of them, given that $\text{rg_t_block_current}$ is 0 when block_to_treat is empty and it belongs to $1..\text{size}(\text{bijection_t_block})$ otherwise.

Proof obligations and B0 check screenshots ATTACH

Component	Type	Checked	POs Generated	Proof Obligations	Proved	Unproved	B0 Checked
 Iter_base	OK		OK	2	2	0	OK
 Iter_main	OK		OK	1	0	1	OK
 Iter_main_i	OK		OK	27	13	14	OK
 Iter_services	OK		OK	8	7	1	OK
 Iter_services_i	OK		OK	8	7	1	OK

Component	TypeChecked	POs Generated	Proof Obligations	Proved	Unreliably Proved	Disproved	Unproved	B0 Checked
block_miniservices	OK	OK	6	6	0	0	0	OK
block_miniservices_i	OK	OK	72	50	0	0	22	OK
block_occupancy	OK	OK	5	5	0	0	0	OK
block_occupancy_i	OK	OK	137	111	0	0	26	OK
block_occupancy_it	OK	OK	8	8	0	0	0	OK
block_occupancy_it_i	OK	OK	16	8	0	0	8	OK
block_occupancy_seq	OK	OK	0	0	0	0	0	OK
block_occupancy_seq_i	OK	OK	6	6	0	0	0	OK
block_services	OK	OK	6	6	0	0	0	OK
block_services_i	OK	OK	14	14	0	0	0	OK
configuration	OK	OK	12	11	0	0	1	OK
inputs	OK	OK	0	0	0	0	0	OK
main	OK	OK	0	0	0	0	0	OK
main_i	OK	OK	0	0	0	0	0	OK