Lecture 8 - January 29

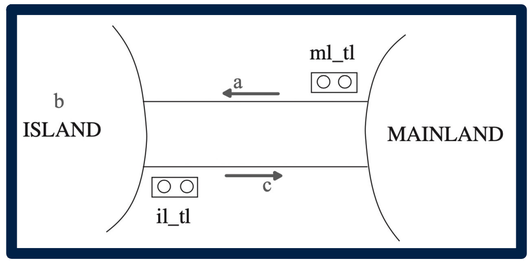
Lab1 Review (Part 2)

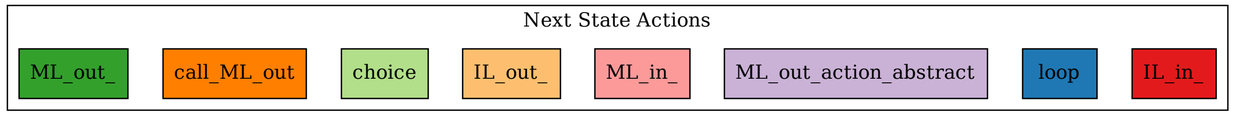
Identifying Atomicity in State Graph
Recall (from EECS3342): System Variant
Encoding & Checking Variant in TLA+

Announcements/Reminders

- **Lab1** solution released
- **Lab2** released
- **TA contact information** (on-demand for labs) on eClass
- Office Hours: 3pm to 4pm, Mon/Tue/Wed/Thu

bridgeController_m1_no_variant.tla





Multiple Labels for Procedure Actions

```
--algorithm bridgeController_m1 {
  variable n = 0, a = 0, b = 0, c = 0;

  procedure ML_out() {
    ML_out_action_abstract: n := n + 1;
    ML_out_action_concrete: a := a + 1;
    return;
  }
```


TLC Errors

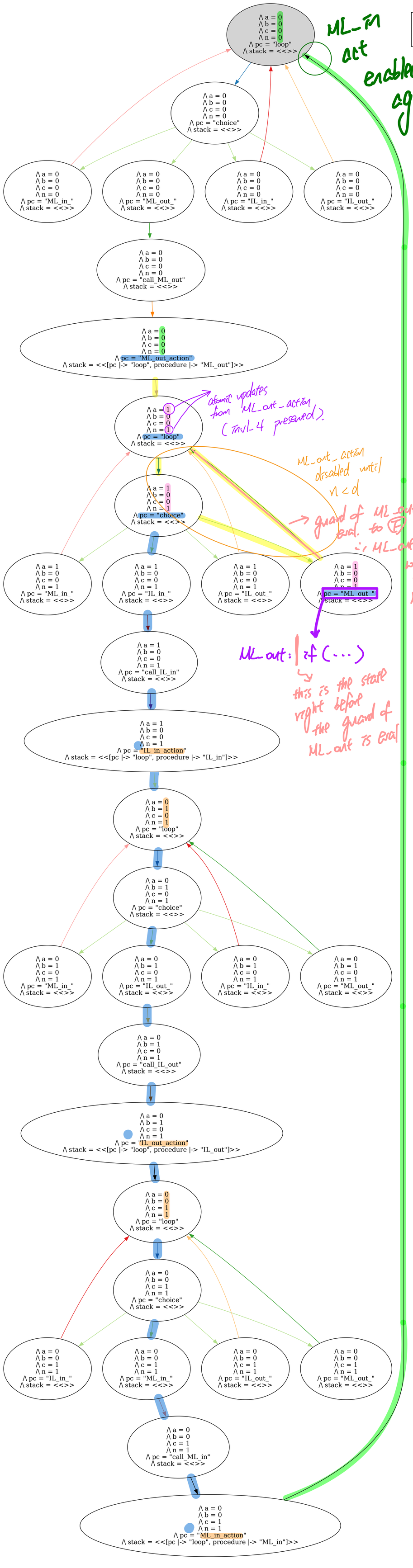
Model_1_d1

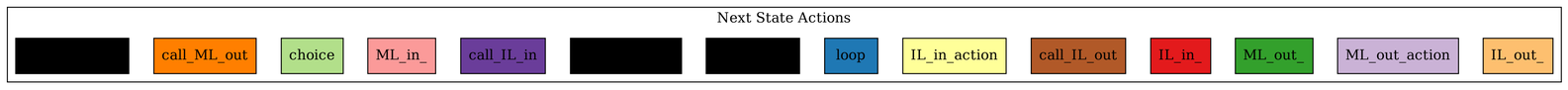
Invariant inv1_4 is violated.

Error-Trace Exploration

Error-Trace

Name	Value
<Initial predicate>	State (num = 1)
<loop line 114, col 9 to line 116, col 44 of module brid	State (num = 2)
<choice line 118, col 11 to line 123, col 46 of module	State (num = 3)
<ML_out_ line 125, col 12 to line 129, col 47 of modul	State (num = 4)
<call_ML_out line 131, col 16 to line 136, col 44 of mo	State (num = 5)
<ML_out_action_abstract line 74, col 27 to line 77, col	State (num = 6)
a	0
b	0
c	0
n	1
pc	"ML_out_action_concrete"
stack	<<[pc |-> "loop", procedure |-> "ML_out"]>>





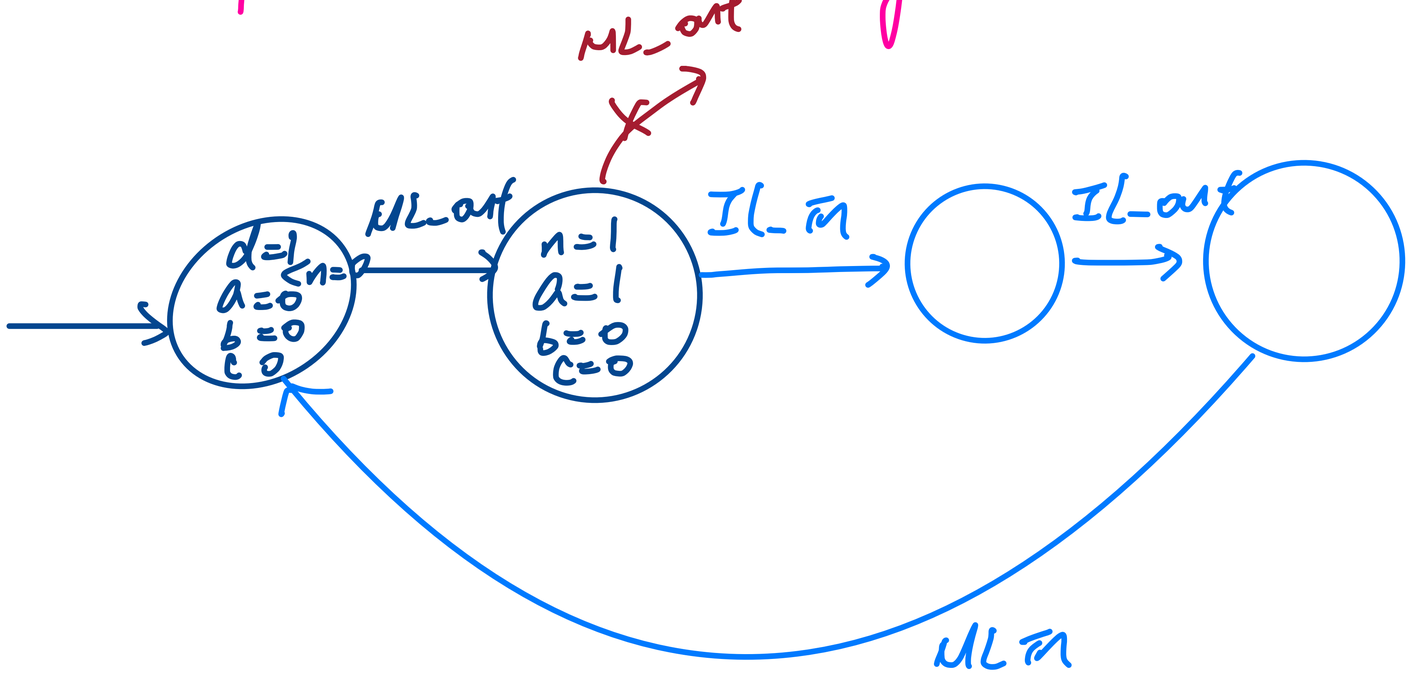
Single Labels for Procedure Actions

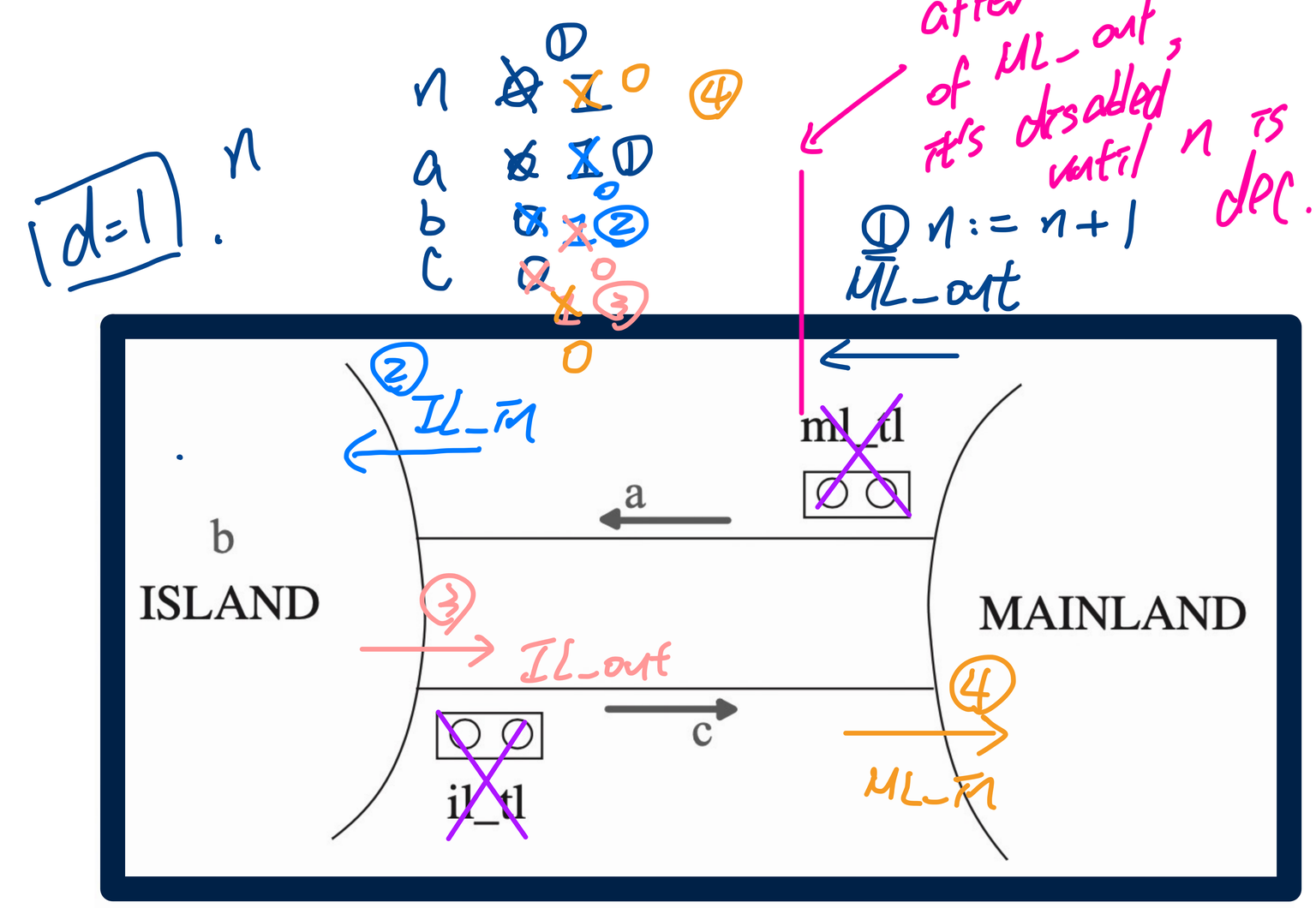
```
--algorithm bridgeController_m1 {
  variable n = 0, a = 0, b = 0, c = 0;

  procedure ML_out() {
    ML_out_action abstract: n := n + 1;
                            a := a + 1;
                            return;
  }
```

bridgeController_m1_no_variant.tla

Exercise

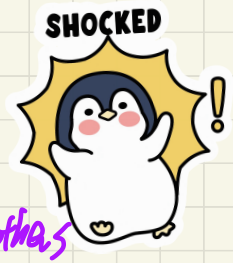
1. Gen. RG d = 2
2. Conceptual sketch of how ML_out is enabled/disabled
3. Inspect the RG and verify that.





system's progress is "locked" in an active way

Livelock Caused by New Events Diverging



↳ certain events happening indef., preventing others from happening.

An alternative m1 (for demonstration)

constants: d

axioms:
axm0_1 : $d \in \mathbb{N}$
axm0_2 : $d > 0$

variables: a, b, c

invariants:
inv1_1 : $a \in \mathbb{Z}$
inv1_2 : $b \in \mathbb{Z}$
inv1_3 : $c \in \mathbb{Z}$

old events

ML_out
when
$a + b < d$
$c = 0$
then
$a := a + 1$
end

ML_in
when
$c > 0$
then
$c := c - 1$
end

new events

IL_in
begin
$a := a - 1$
$b := b + 1$
end

IL_out
begin
$b := b - 1$
$c := c + 1$
end

(unconditional)

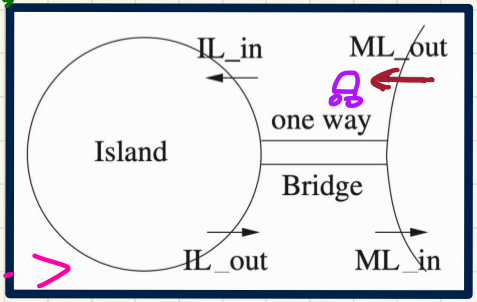
solution: have a measure on imposing an upper bound on the # of interleavings of new events

while(1) {

} no

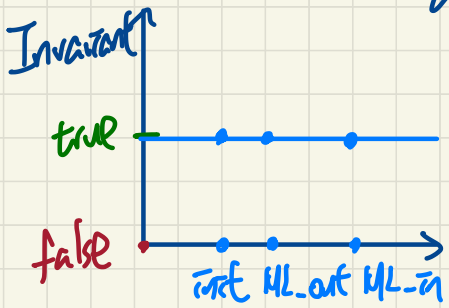
Abstract transitions: <init, ML_out, skip, skip, skip, skip, ... >
m1
new events occurring indefinitely.

Concrete transitions: <init, ML_out, IL_in, IL_out, IL_in, IL_out, ... >
a possible trace that's problematic.

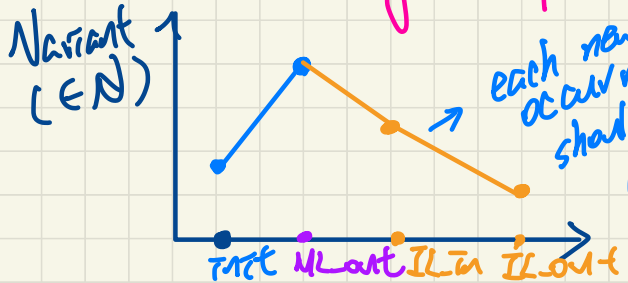


Invariant vs. Variant

Invariant : Boolean expression that should always hold (after each event occurrence)



Variant : Integer expression that may change after event occurrence.



each new event occurrence should strictly decrease the value of variant ⇒ # new events in interleaving is finite ⇒ no livelock

Use of a Variant to Measure New Events Converging

fixed

variables: a, b, c

invariants:
- inv1_1 : $a \in \mathbb{N}$
- inv1_2 : $b \in \mathbb{N}$
- inv1_3 : $c \in \mathbb{N}$
- inv1_4 : $a + b + c = n$
- inv1_5 : $a = 0 \vee c = 0$

ML_out
when
$a + b < d$
$c = 0$
then
$a := a + 1$
end

ML_in
when
$c > 0$
then
$c := c - 1$
end

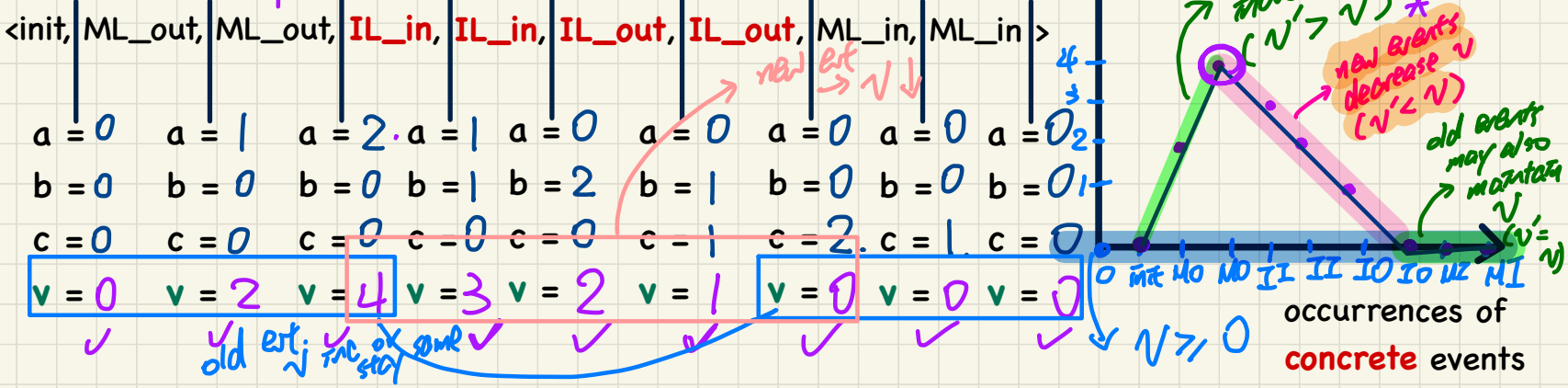
IL_in
when
$a > 0$
then
$a := a - 1$
$b := b + 1$
end

IL_out
when
$b > 0$
$a = 0$
then
$b := b - 1$
$c := c + 1$
end

* Given that the starting value of V of the first new event is finite,

Variants for New Events: 2 · a + b

the # of interleaved new events is also finite.



PO of Convergence/Non-Divergence/Livelock Freedom

Variants for New Events: 2 · a + b

Variant Stays Non-Negative

$A(c)$
$I(c, v)$
$J(c, v, w)$
$H(c, w)$
$\vdash$ $V(c, w) \in \mathbb{N}$

guard of new event

NAT

IL_in/NAT

new event

$v \geq 0$

$d \in \mathbb{N}$
$d > 0$
$n \in \mathbb{N}$ $n \leq d$

$a \in \mathbb{N}$ $b \in \mathbb{N}$ $c \in \mathbb{N}$ $a > 0$ $a = c \vee c = 0$ $a + b + c = n$

$\vdash 2 \cdot a + b \in \mathbb{N}$

* $v' < v$ $(b+1)$

$2 \cdot a' + b' < 2 \cdot a + b$

$(a-1)$

variant: V(c, w)

v

$V(c, w)$

v

$V(c, F(c, w))$

IL_in/NAT

new event

occurrences of **new** events

A New Event Occurrence Decreases Variant

$A(c)$
$I(c, v)$
$J(c, v, w)$
$H(c, w)$
$\vdash$ $V(c, F(c, w)) < V(c, w)$

$a' = a - 1$ $\wedge$ $b' = b + 1$

effect of new event

v v

VAR

IL_in/VAR

new event

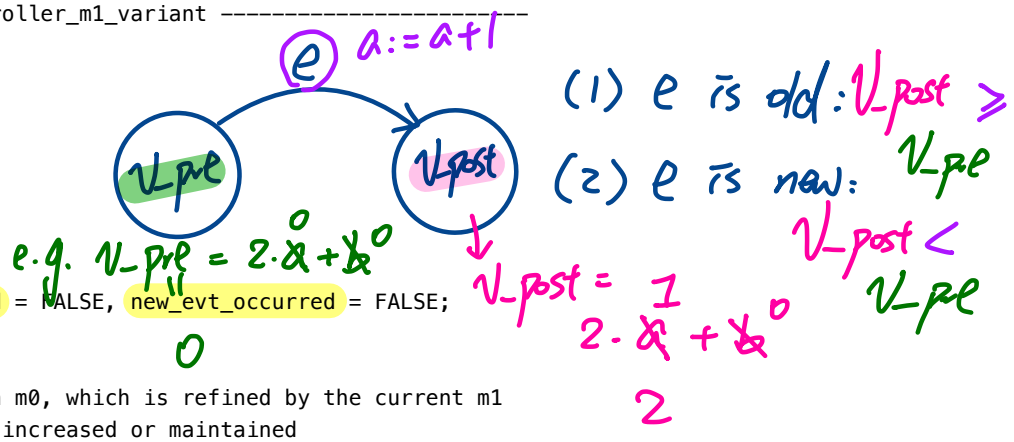
v's value strictly decreased.

$d \in \mathbb{N}$
$d > 0$
$n \in \mathbb{N}$ $n \leq d$

$a \in \mathbb{N}$ $b \in \mathbb{N}$ $c \in \mathbb{N}$ $a > 0$ $a = c \vee c = 0$ $a + b + c = n$

* $\vdash 2 \cdot (a-1) + (b+1) < 2 \cdot a + b$

```
------------------------ MODULE bridgeController_m1_variant ------------------------
EXTENDS Integers, Naturals, Sequences, TLC
CONSTANT d
ASSUME /\ d \in Nat
       /\ d > 0
(*
--algorithm bridgeController_m1 {
  variable
    n = 0, a = 0, b = 0, c = 0,
    V_pre = 0, V_post = 0, old_evt_occurred = FALSE, new_evt_occurred = FALSE;

  (*
     Old events: ones that already exist in m0, which is refined by the current m1
     Value of the system variant is always increased or maintained
       by each occurrence of an old event.
  *)
  procedure ML_out() {
    ML_out_action: n := n + 1;
                   a := a + 1;
                   return;
  }

  procedure ML_in() {
    ML_in_action: n := n - 1;
                  c := c - 1;
                  return;
  }

  (*
     New events: ones that do not exist in m0, which is refined by the current m1
     Value of the system variant is always decreased
       by each occurrence of a new event event.
  *)
  procedure IL_in() {
    IL_in_action: a := a - 1;
                  b := b + 1;
                  return;
  }

  procedure IL_out() {
    IL_out_action: b := b - 1;
                   c := c + 1;
                   return;
  }

  {
    loop: while (TRUE) {
      (* Without the first two updates resetting the event log,
         when new_evt_occurred == true, after the next line,
         V_pre == V_post, which will violate the VAR variant constraint.
      *)
      update_variant_pre: new_evt_occurred := FALSE;
                          old_evt_occurred := FALSE;
                          V_pre := 2 * a + b;
      choice: either {
                ML_out: if ( (n < d) /\ (a + b < d) /\ (c = 0)) {
                  call ML_out: call ML_out();
                  update_evt_log_ml_out: new_evt_occurred := FALSE;
                                         old_evt_occurred := TRUE;
                                         V_post := 2 * a + b;
                };
              }
              or {
```



captures the pre-state value of V for the next event
↳ old or new.

atomic updates

```
      ML_in: if ( (n > 0) /\ (c > 0) ) {
        call_ML_in: call ML_in();
        update_evt_log_ml_in: new_evt_occurred := FALSE;
                              old_evt_occurred := TRUE;
                              V_post := 2 * a + b;
      };
    }
    or {
      IL_in: if ( a > 0 ) {
        call_IL_in: call IL_in();
        update_evt_log_il_in: new_evt_occurred := TRUE;
                              old_evt_occurred := FALSE;
                              V_post := 2 * a + b;
      };
    }
    or {
      IL_out: if ( (b > 0) /\ (a = 0) ) {
        call_IL_out: call IL_out();
        update_evt_log_il_out: new_evt_occurred := TRUE;
                               old_evt_occurred := FALSE;
                               V_post := 2 * a + b;
      };
    };
  }
 }
}
*)
\* BEGIN TRANSLATION (chksum(pcal) = "ce02e87c" /\ chksum(tla) = "5f2f5c21")
...
\* END TRANSLATION

\* checking invariants
inv1_1 == a \in Nat
inv1_2 == b \in Nat
inv1_3 == c \in Nat
inv1_4 == a + b + c = n
inv1_5 == (a = 0) \/ (c = 0)

\* checking variants
variants == 2 * a + b >= 0
event_log_consistent == ~(/\ old_evt_occurred = TRUE /\ new_evt_occurred = TRUE)
variant_not_decreased == (old_evt_occurred = TRUE => V_post >= V_pre)
variant_decreased == (new_evt_occurred = TRUE => V_post < V_pre)

\* checking deadlock freedom
guard_ML_out == /\ (n < d)
                /\ (a + b < d)
                /\ (c = 0)
guard_ML_in == /\ (n > 0)
               /\ (c > 0)
guard_IL_in == a > 0
guard_IL_out == /\ (b > 0)
                /\ (a = 0)
deadlockfree == guard_ML_out \/ guard_ML_in \/ guard_IL_in \/ guard_IL_out
================================================================================
```