PRACTICAL TLA: PART I

```
---- MODULE Filename ----
EXTENDS Integers

(*--algorithm algo_name

begin

end algorithm; *)
====
```

PlusCAL is inside comment.

things outside pet ignored (put notes/defs there)

Expression evaluator:
Model → Model checking results → Evaluate constant expression

```
x = y
x /= y
x /\ y      (like ∧ and ∨)  - and, or
x \/ y
~x          not (binary?)
x := y      (PLUSCAL only)
```

TRUE FALSE

raw TLA: = first true ~~initialization~~ initialization

basic values ⟸ string, integer, boolean, model value

= next — comparison (eq. check)

```
1..3 = {1,2,3}
```

(// Python)

EXTENDS Integers → + - * ~~/~~ \div %

constructed types: (sets, structures) (tuples / sequences), functions

1. Sets {"a", "b"} : all of the same type

```
x \in set
x \notin set   <=>  ~(x \in set)
\union   \cup
\intersect   \cap
\   (difference)
Cardinality(set)   <- EXTENDS FiniteSets
```

EXTENDS A, B, C, D

~~filter~~ filters and maps:

```
{x \in 1..2 : x < 2}     (filter)
{x * 2 : x \in 1..2}     (map)
```

2. tuples : don't need to be the same set

```
t = << "a", {1,2} >>
tup[1] = "a"
tup[2] = {1,2}
```

EXTENDS Sequences →
- Head(s)
- Tail
- Append(Seq, el)
- \o (combine)
- Len(s)

③ Structures: map strings to values ~~of the se~~ (possible different types)

[a ↦ 1, b ↦ << 1, {} >>].b

S.field (read)

---

actually structures and sequences are of the same ~~thetype~~ type

⚠ updating structures:

```
variables x = << 1, [a ↦ {}] >>;
begin
  x[2].a := x[2].a \cup {1,2};
end algorithm;
```

---

assert condition ← EXTENDS TLC

skip ← noop (loop / if filler)

```
if cond
  body
elsif c2 then
  body
else
  body
end if;
```

(then)

```
while cond
  body
end while;
```

(do)

MACROS: EXTENDS TLC

```
macro name(a1, a2) begin
  \* assignments
end macro;
```

} before begin ⚠

name(1, 2);

assignments, assertion, ifs ⚠ NOT while loop!

```
macro set_false(x) to
  x := FALSE;
end macro;
```

variables BEFORE macros BEFORE begin (algo)

---

initialization of var:

```
variables x \in {1, 2, 10};
```

← x will be any of these

BOOLEAN (means {TRUE, FALSE})

UNION {s1, s2, s3}

SUBSET set

s1 \X s2 (cartesian product → set of all tuples)

a \X b \X c ← set of all triples a × b × c

⚠ ~~a~~ a \X (b \X c)

? set of structures:

[key : [val set]] ← set of xs such that x is a structure where the value of key \in [val set]

[a : {"a", "b"}] ⇒ {[a : "a"], [a : "b"]}

if you want some key to be always equal one value v, do [key : {v}, k2 : set2]

---

error:

TLC threw an unexpected exception
...
... Assert(t, ...)
the first argument evaluated to FALSE

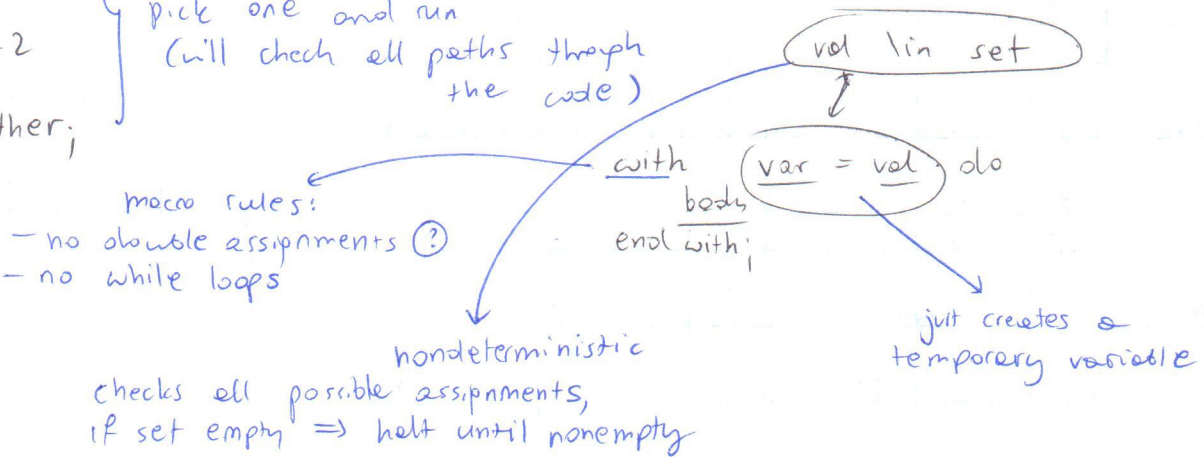
⇒ this means failure of assertion (assert)

---

nondeterministic behavior: EITHER, WITH

either
  branch 1
or
  branch 2
:
end either;

pick one and run (will check all paths through the code)



! with gives values, not references: with t \in {x, y} do t := 1; end with;

will not write to neither t, nor y

1T for translating PlusCAL to TLA+.

TLC reports the first error (assertion error) it finds and stops.

liveness bug := ? arriving in order is satisfied if messages never arrive

deadlocks.

CHAPTER 3: Operators & functions

OpName(a1, a2) == expr operators are like procedures. defined with ==
called OpName(1, 1+3) or OpName

✗ proper leverage override overwrite
no () →

SetOfFour(set) == set \X set \X set \X set
BinTypes == {"trash", "recycle"} ← no ;

if you define some constant as operator (a == 10) then it can't be accidently overriden
BEFORE (* algorithm !

TLA+ doesn't use ;; it is not whitespace sensitive, PlusCal is !

if you want to put operators in PlusCal, you have to use define

define ← variables Definitions → Macros ~~THE~~ THE RIGHT ORDER
  Na == a ≥ 0 ∧ b ≤ 0
end define;

ok for formatting
the only place where whitespace matters

∧ A
∧ B
∨ C
∧ D
→ A ∧ (B ∨ C) ∧ D

hipher order operators: (arity must be specified)

Add(a,b) == a + b
Apply(op(_,_), x, y) == op(x, y) // Apply(Add, 1, 2)

LAMBDAs (only ~~for~~ as arguments !)

Apply(LAMBDA x, y : x + y, 1, 2)

!, \o ... ten operators, see User Definable Operator Symbols (Help → The TLA+ Cheat sheet)

Set ++ el == set \cup {el}
Set -- el == set \ {el}

{1,2} --2

Invariant :: a boolean expr evaluated ~~after~~ after each step
(if you make them evaluate in Model)
should they ever fail → computation stops

invariants > assertions ← errors nicer
← halts immediately, not at the end of run

create dedicated operators for invariants (you don't, have to)
~~inline~~ boolean exprs

\A x \in set : P(x)
\E x \in set : P(x)
~\A
~\E

\A x, y \in set :
\A x \in set, y \in set1 :
\A <<x, y>> \in S1 \X S2 :

P => Q P <=> Q Xor(a, b) == ~a <=> b

∧ p
∧ q
⇒ r
} (p ∧ q) ⇒ r

∧ p
∧ q
⇒ r
} p ∧ (q ⇒ r)

EXPRESSIONS (no ;)

LET
  a == p
  b == q
IN expr

TLA+ expression, PlusCal's if is a statement

IF test THEN t ELSE f

x := IF p THEN 1 ELSE 0

CASE x = 1 → TRUE
 [] x = 2 → TRUE
 [] x = 3 → 10
 [] OTHER → FALSE

always true. you can ommit OTHER, but then no match ⇒ an error.

! more than one proper cases matches ⇒ undefined behavior

CHOOSE x \in S: P(x)

selects if such x that P(x) is true
(it chooses arbitrary x, it doesn't check all)
none ok → error

IndexOf(seq, el) ==
  CHOOSE i \in 1..Len(seq):
    seq[i] = el
IndexOf(<< 1, 8, 7 >>, 7) ↦ 3

probably $\Theta(n^2)$ ☹

Max(S) == CHOOSE x \in S: ∀ y \in S: x ≥ y
Max(1..10)

CHOOSE << x, y >> ∈ (-10..10) × (10..10):
  ∧ 2x + y = -2
  ∧ 3x - 2y = 11
↦ << 1, -4 >>

FUNCTIONS

[x ∈ S ↦ P(x)]
[x ∈ S, y ∈ S' ↦ << x, y >>]
[x, y ∈ S ↦ P(x, y)]

f[arg] ← Mathematica-like calling

tuple := function with domain 1..n
struct := function with domain as a set of strings

[x ∈ 1..2 ↦ 2*x]

f[a, b] ⟺ f[<< a, b >>] equivalent

(p. 53 why 3 states?)

operators called with (), functions with []
[Op == [x \in S ↦ P(x)]
  Op[x \in S] == P(x)
both valid if op. takes 0 arguments

M(s, n) == [x \in S ↦ n]
only this syntax valid if function depends on operator's args

operators work on arbitrary inputs
functions have specified domain

useful library!

```
PT == INSTANCE PT
SumTo(n) ==
  PT!ReduceSet(LAMBDA x, y: x + y, 0..n, 0)
```

(you need EXTENDS Integers for 1..10 to work)

FM → RUN model

DOMAIN fun gives all possible input to function fun
DOMAIN <<0, 1, 70>> → 1..3
DOMAIN [x ∈ BOOLEAN] == x → [TRUE, FALSE]

EXTENDS TLC
f1 @@ f2 merges functions, like

merge(f, g) == [
  x ∈ (DOMAIN f) ∪ (DOMAIN g)
  ↦ if x ∈ D_f then f[x] else g[x]
]

EXTENDS TLC
a :> b ⟺ [x ∈ {a} ↦ b]
(2 :> 7)[2] → 3
("a" :> "b").a → "b"

Set of all functions from D to D'
[D -> D'] → not ↦

n-element sequence of S: s ∈ [1..n → S]

// {config ∈ [Flags → BOOLEAN]: ∃f ∈ Flags: config[f]}
set of all configurations of flags, such that at least one is set

PT!ReduceSet(LAMBDA x, acc: x + acc, Items, 0)

CONSTANTS, MODELS, IMPORTS

constants := values defined in model, not in specification

~~Consts~~

CONSTANTS a, b, Abbco, Defa — cannot be modified
set in

Model → model overview → what's the model

model value := if you assign model value to a constant, it becomes a new type, equal only to itself

\subseteq

Items ← [model value] {a, b, c}
Items ← [model value] <Symmetrical> {a, b, c}

} you set this in model overview

if your core is symetrical then this can be significantly less work

In spec you can specify the constraints of constants

ASSUME A \subseteq 1..n
ASSUME B \in C
ASSUME C > 10

can use constants, constant operators, but nothing other than ~~variable~~ constant

EXTENDS Integers, Naturals

Int = {..., -1, 0, 1, ...}
Nat == {0, 1, 2, ...}

} infinite sets, can only test membership

ASSUME A \in Nat
ASSUME B \in Nat \ {0}
ASSUME C \subseteq Int

Help → Table of Contents

You almost always want Temporal formulas, not No behavior spec (for Cloud3 constant expressions and not anything else)

WHAT to check:
Invariants, Properties

How to run → optimizations

Advanced options

Additional definitions → eg. F(x) == x^2
C ← F(1)
useful for complex ~~model~~ setups defining addl. operators

State constraint → if some state doesn't satisfy it, skip it (no error, no inv. checking, no future states)

definition override → custom intervals, eg. ~~F(x,y) ← 3~~
+(x, y) ← 3

Some people do
with x \in Int do
...
end with;
and set Int ← 0..1000
to clarify intent, but still make it runnable

TLC options → by default BFS model-checking mode
simulation mode → random trees instead of search, use search for smaller spaces, use simulation for larger ones (to be sure)

Error-trace exploration (between error output and error trace)

x - value at the beg. of a step
x' - value at the end of a step
op(x,y)' - the value of op(x,y) at the end of a step

JavaTime — current epoch

Print(to-print, ret-val) prints both, returns ret-val
PrintT(val) == Print(val, TRUE)

Assert(3 > 5, "3 > 5, bad")

(this is all TLA, PlusCAL's assert is defined in terms of this Assert)

LET x == 5 y == 6 IN ... ok

ToString(5) \o " is more than " \o ToString(3) concatenation?

SortSeq(seq, Op(_,_)) reversed sort
SortSeq(<< 1, 3 >>, LAMBDA x,y: x > y)
Permutations(set)

Specification can have multiple modules
library path OR the same dir as your spec

additional module is only for operators, values, data structures, but no PLUSCAL ?

EXTENDS A, B, C, D
INSTANCE A ← only 1 at a time

LOCAL ... ← if this is in module, the proceeding def. will not be imported when importing that module
LOCAL INSTANCE A

EXTENDS
only one per file allowed (but can take multiple args)
dumps everything into namespace
the module can't have any constants (?)

INSTANCE
one imported lib per statement (no INSTANCE A, B, C ...)
you can namespace modules PT == INSTANCE PT
then call operators: PT!OpName
you can import parametrized modules (constants def. at import time)
INSTANCE M WITH Const1 ← v1, Const2 ← v2
M == INSTANCE M WITH C1 ← v1, C2 ← v2

without this, dumps into namespace

if you INSTANTIATE a module without defining constants, they'll default to values of consts/operators with the same names.

x == 1
y == 2
INSTANCE M * has x, y as constants

CH5: CONCURRENCY

label := a point of atomicity. TLC executes everything in a single label atomically
a label (=) ~~but~~ simple action

// TLA vs TLC (?) vs TLA+

pc variable (product of translating PlusCal → TLA+)
pc = "A" then next state is label A

goto LabelName surprisingly useful since specs are smaller than programs

"Done" - end of all processes, you can set your label to it, but u can goto "Done"

label placement rules: (MUST)
- each process
- every while
- after every goto
- after either or if
  ↑ any branch has a label

NO labels
- inside a macro or with statement
- using the same label twice
- (!) assigning to the same variable twice inside a simple label

Lab:
x := 1;
y := 2; ← error — this y should be x ? (?)

Usually we write sth. like begin Label: ...

Lab2:
either x := 1;
or x := 2;
end either; } ok

Each process must be assigned to value

With processes, pc is no longer a value, it's a function representing pc of each process

Invalid:
str.k1 = 1;
str.k2 = 2; | error (str)

Valid:
str.k1 = 1 || str.k2 = 2; | ok

new special operator (simultaneous assignment) for this purpose.

```
process reader = "reader"
variables curr = "none";
begin Read:
  while TRUE do
    curr := Head(queue);
    queue := Tail(queue);
  end while;
end process;
```

processes local variable

await queue /= <<>>;

await ~~expr~~ expr or when expr
prevents a step from running until expr

(?) putting await after an action → ~~discouraged~~ discouraged
→ weird (p. 893 | 100)

useful pattern: ("possibly")
```
either skip;
or Label:
   action;
end either;
```

// Lumen database !

- deadlock can happen if $S = \emptyset$

  with $s \in S$;
- you can disable deadlock check in Model → Overview → What to check

! change style process to multiple processes:

process reader = "r" → process reader ∈ {"r1", "r2"} !

self ← value of the current process

! all values for processes must be comparable, eg. either all strings or ~~all ints...~~ a set of model values ! or all ints

! or some strings, and some model vars

```
procedure name(arg=def) begin
  Label:
    ==
    return;
end procedure;
```

→ call name(arg1)

procedure - is like a macro, but you can put labels in it
- can have process-like local variables (=), but not (∈)
- return ends the procedure, not passes information ! ← ? what?

! a call to procedure must be immediately followed with
- goto
- a label
- the end of a block
- return

! procedures must be defined after macros and before processes
(procedures can use macros
macros can't use procedures)
(procedures can use processes, but the inverse is true)

! self can still be used inside procedures or macros (value of the calling process)

you can formalize many processes, not just algorithms with processes.
eg. simulations of concurrent behaviours.

---

both versions are correct: (...5)

CONSTANT A, B — CONSTANTS A, B

variable a=1, b=2; — variables a=1, b=2

```
variables
  a = 1,
  b = 2,
  c = 3;
```

~bool_val

CH 6: TEMPORAL LOGIC

examples of temporal properties:
does the algo always terminate?
will all the messages get processed?
if disrupted, will the system return to stability over time?
is the DB eventually consistent?

model overview → what to check → properties → Termination

Stuttering:=
TLC usually ~~either~~ select one of the available labels at each time and runs it, but sometimes it just takes no action.

to this point it didn't matter (for ~~safety~~ (safety checks)) ~~if~~ valid state + stutter → still valid state.
but for liveness checks (one that verifies if some state is reached eventually)
if you never finish you failed.

Stuttering can represent ~~or~~ request timed out
server crashing
awaited signal never coming

It has to be always enabled, else no guarantees

weak fairness (⇐) fair process
weakly fair action, if it stays enabled (always) will eventually happen

strong fairness (⇐) fair+ process
strongly fair action, if repeatedly enabled will eventually happen

strong fairness is used rarely in specs since it's a costly assumption

A:+
action → makes this label weakly fair
(if ~~already~~ weakly fair) makes this label strongly fair

--algorithm ↷ --fair algorithm make the whole algo weakly fair

[]Property - always, Property always holds. Rarely used (use invariants instead)
~[]Property in at least 1 ~~if~~ state Property does not hold
~<>P ⇔ []~P P is never true

Ctrl + Enter accepts most prompts.

! notice that in ~~a~~ pereveted PlusCal ~~translation~~ translation:

Termination == <>(∀ self ∈ ProcSet: pc[self] = "Done")
temporal properties are just ~~ob~~ operators, in PlusCal you can define them with define and add them in model.

currently, you can't check ∈ in temporal properties (no <>(x ∈ S))

P ~> Q leads to. if in state #i P is true, then in some state #j, j≥i Q is true.

P ~> []Q P leads to Q becomes true forever sometime in the future

[]<> P P is always (eventually true)
<>[] P P is eventually (always true)

!? for finite specs it's the same — P true @ termination

(for every state #i there ∃ j≥i s.t.: P holds @ #j
so for the last one also)

for infinite specs ~~it / prover that~~
<>[] ∃ i ∀ j≥i P holds @ #j (...0000100011001100111(1))
[]<> ∀ i ∃ j≥i P holds @ #i (~~is ep~~ 0100000101001...)

~~infinite spec~~ []<>P ⇔ (~P ~> P) !

Checking liveness is very slow
invariants checked on each state seperately (parallelizable)
temp. props. checked on series of states (different algos, no parallelization)

- place temporal ~~prop~~ props. in different model so you can check invariants quicker by themselves
- check liveness on smaller models and invariats on bigger ones

---

! DO NOT MIX TEMP. PROPS WITH SYMMETRY SETS ! (skipping identical results skipping ≠ liveness checks sometimes)
THIS MAY LEAD TO FALSE NEGATIVES !
(usually TLC has only false positives for errors)
TLC WILL WARN YOU IF YOU DO THIS.

// at most one ~~be~~ is in critical section: ~~be~~ pretty

∀ t, t' ∈ Threads: t ≠ t' ⇒ ~(~~t~~ pc[t] = "cs" ∧ pc[t'] = "cs")

you actually can check thing with ∀ x ∈ S in temp. properties if S is a constant!

---

livelock := threads ~~for~~ locked cycling for ever →

path p. 108 | 125 you can move ~~the~~ code of some process to a procedure and create both fair & unfair versions with

fair process p = 1
begin FP:
call proc();
end process;

F11 runs model checking

! self is only defined for sets of processes, so ~~if you~~
if you use it, do
p ∈ in 1..N instead of p = 1 !

PRACTICAL TLA+ PART II

CH 7: Algorithms

most algorithms:

Expected(input) = ...
(x-algo ==>

(algo)
assert out == Expected(input)

PT! SeqOf(set, len)

*)

performance of algorithms

click New Model (after) you compiled PlusCal
so that you get correct things set up in Model
(else set TemporalFormula to Spec)

! you can override operators !

```
n ** m ==
  LET f[x ∈ 0..n] ==
    IF x = 0 THEN 1 ELSE m * f[x-1]
  IN f[n]
```

7**5 → 16807

counter variable and assert, eg $2^{(counter-1)} \leq Len(seq)$ so it fits

Advanced options → TLC options → Cardinality of largest enumerable set
(10^6 by default)

this is asymptotic for worst-case scenario.
average & best-case: TLC will not help you usually

10 \div 7

(l+h) \div 2 → with a = l+h; b = a \div 2; do assert a <= MaxInt (your own)

! this is how you check for overflows

l + (l-h) \div 2 doesn't overflow

or

```
NoOverflowsInvariant ==
  ∀ x ∈ myVars : x ≤ maxInt
```

CH 8: Data Structures

data structures should be written as separate modules, then imported

LOCAL INSTANCE TLC ← if you import f, TLC will not be imported also

If you have some parsing error, check == vs =

writing CHOOSE x ∈ MyDomain : weird_property(x) is a good thing to explore the space

! Inotin !

tip: IF ring(LL):
choose n∈N : True
else choose n∈N : n ∉ Range(LL)
⟺ choose n∈N : ~ring(LL) ⇒ n∉range(LL)

LOCAL OP == Def) operator local to the module

you generally define some predicates that test properties and then you use them in filters on operators that define all data structures to get interesting subsets.
you do this in separate module (elegance), declaring some operators local
you define sanity checks and validation in separate (main) module

! if you have no algorithm (pure TLA+) you want No Behavior spec

Assert(cond, << "Cond not met", counterexample:", element >>)

and run Valid in evaluate Count expression where
Valid == ∀ e ∈ D : P(e)

CH 9: State machines

? how does either await P1 or await P2 end either; (either + await) work?
eg theo
Picks only the ones that are possible!

(?) await is an instruction that either
(~) does nothing (condition not) or if a single atomic blocking step (passes execution somewhere else)

state machines usually look like

```
variables (shared)
process state 1
begin P1:
    await state 1 - check
    either
        await p1_t1
        [transition 1]
    or
        transition 2;
```

* possible transitions, each awaits preconditions

{ scaffolding

TLA+ (not PlusCal) has some form of refining (specializing) specs
but not described in a book

---

most infinite loops:

```
Label:
    ...
    goto Label;
```

(or while TRUE do ...)

---

this book gives you # of states and you can see if you got the spec right, good idea

auxiliary) ghost data := additional data used to track invariants in the spec

CH 10: Business Logic

```
set ++ x == set \cup {x}
set -- x == set \ {x}
```

{1} \subseteq {1, 3}

```
\* END TRANSLATION
TypeInvariant ==
    ∧ library ∈ [Books → NumCopies ++ 0]
    ∧ books   ∈ [People → SUBSET Books]
```

you can access private variable here (post translation one)

} type invariant pattern

TLC Model Checker → Clone Model → (Rich your model) !

library simulation turns out quite hard to do (whole chapter)

advanced options → state constraint → ∀ p ∈ People : # books[p] ≤ 1
(limiting space to only these states)

CH 11: MapReduce

first, you might hard code space size (eg. 4 books in a library), then end have fast responses, then generalize

define operators that perform actions correctly in a functional way

NULL is often defined as a special ~~concept~~ model value.

while loop must come directly after a label!

PT! OrderSet(set) will order set, but TLC will not try different orderings! so specs using it might potentially break using different ordering

a %% 3

PT! Seq Mod ?

Reducer must be fair (else we can't guarantee anything)
there must be at least 1 fair worker
it doesn't matter which worker is fair (CHOOSE, and reduction in space)
reducer might not detect an unfair worker failing
-||- must never decide that the fair worker failed (fairly decide)

if you need some processes fair and some unfair, extract body into procedure and run that procedure in different processes

---

put comments explaining either branches: either * Pos 1
or * Pos 2

// p. 180

[a ↦ a, b ↦ b] is ok, if [“a” ↦ a, “b” ↦ b]

C syntax is better (less visual clutter)
process (w = 10) .C1
while ( ) < > { }

---

invariant using private vars → after translation
invariant using also global vars → define statement

```
ReduceSet(op(_,_), set, acc) ==
  LET f[s ∈ SUBSET set] ==
    IF s = {} THEN acc
    ELSE LET x == CHOOSE x ∈ s : TRUE
         IN op(x, f[s \ {x}])
  IN f[set]
Range(f) == {f[x] : x ∈ DOMAIN f}

OrderSet(set) == CHOOSE seq ∈ [1..Cardinality(set) → set] : Range(seq) = set
Preimage(f, s) == {x ∈ Domain f : f[x] ∈ s}

TupleOf(set, n) == [1..n → set]
SeqOf(set, n) == UNION {TupleOf(set, m) : m ∈ 0..n}
ReduceSeq(op(_,_), seq, acc) ==
  ReduceSet(λ i,a : op(seq[i], a), DOMAIN seq, acc)
```

— APP. C: PlusCAL → TLA+ —

book „Specifying Systems"
hyperbook

modal logic
↳ temporal logic …
- □P always true
- ◇P sometimes true

behavior := sequence of states

(?) next-state relations

action := predicate between two states, ex. x ∈ vars P
then x - initial state of x, x' - next state

Increases == x' > x
DecreaseFromAtLeast3 == ∧ x ≥ 3
∧ x' < x

Counter modulo 3:

Inc == x < 2 ∧ x' = x+1
Reset == x = 2 ∧ x' = 0
Init = (x = 0)
Next = Inc ∨ Reset
Spec = Init ∧ □ Next

```
variables x = 0;
{
  while (TRUE) {
    either {
      await x < 2;
      x := x+1;
    } or {
      await x = 2;
      x := 0;
    }
  }
}
```

---

in spec, all next (var') must be specified

you also need to make it stuttering-invariant
(at each between each 2 steps you might add another empty step — needed for combining executions)
to do this

□Next → □[Next]_var
either Next or var unchanged

(do vars = << x, y, z, … >>
□[Next]_vars)

TLA := Temporal Logic of Actions

seq (+) e == Append(seq, e)

Avoid mixing ~~state~~ model and spec constraints
(makes sense) (no errors)