TLA+ 101

Specifying Concurrent Processes in PlusCal

Links

Download and install TLA+ Toolbox:https://lamport.azurewebsites.net/tla/toolbox.html

- You'll need Java: https://java.com/

Command Line Interface (Hacky!): https://github.com/pmer/tla-bin

Workshop materials: https://github.com/salmans/pluscal-workshop

References

- Specifying Systems: https://lamport.azurewebsites.net/tla/book-02-08-08.pdf
- PlusCal User Manual: https://lamport.azurewebsites.net/tla/p-manual.pdf
- PlusCal Online Tutorial: https://learntla.com/introduction/
- Summary of TLA+: https://lamport.azurewebsites.net/tla/summary.pdf

Introduction

TLA+ is a specification language for modeling and analyzing systems:

- It models “software above the code level/hardware above the circuit level”
- Models system execution as a *state transition system*.
- Applies *temporal* reasoning to analyze *all* possible executions of the system.
- Created by Leslie Lamport (he also created LaTeX).

Terminology

- *TLA+:* a declarative logical language for describing finite state machines.
- *TLC:* a model-checker for analyzing specifications written in TLA+.
- *PlusCal:* an *entry-level* pseudo-language for writing (concurrent/distributed) algorithms.

PlusCal

- It offers constructs for describing concurrency and non-determinacy.
- TLA+ expressions can be used in PlusCal.
- It gets translated to a TLA+ specification.

This Workshop

Part 1: maze solver

- Introduction to the TLA+ Toolbox
- Review basic features of PlusCal
- Write simple TLA+ expressions

Part 2: simple distributed CRDT system

- Model communicating processes
- Verify eventual consistency properties
- Model network properties

PlusCal Basics (1)

Click on `File > Open Spec > Add New Spec…` to create a new TLA+ specification. Call it "`maze.tla`".

- **Warning:** file names must be the same as their TLA+ module names. We use "`maze`" in our examples throughout the first part of the workshop.

Copy and paste the code template into your specification:

- `EXTENDS` imports a number of useful modules. We'll need these modules for pretty much everything!
- PlusCal is written inside a TLA+ block comment between `(*` and `*)`, starting with `--algorithm`.
- Variable declarations must come first.
- `Start` is a ***label*** for the initial *step* (state transition). Under a label, a variable can be assigned **at most once**.

Save your specification. Click on `File > Translate PlusCal Algorithm (cmd + t)`.

- Check out the TLA+ translation at the bottom.

```
------------------------------ MODULE maze ------------------------------
EXTENDS TLC, Integers, Sequences, FiniteSets

(* --algorithm
begin
variables x = 4, y = 1;

\* PlusCal algorithm:
begin Start:
x := 3;  \* temporarily here to make things work

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

Models and Invariants

Click on `TLC Model Checker > New Model...` to create a new model. Give it a name like "`test_model`".

- **Warning:** throughout this workshop, make sure that the behavior of the model (under `“What is the behavior spec?”`) is set to `“Temporal formula”` with `Spec` as its parameter. When you overwrite the entire specification file, the tool may reset this option to `“No behavior spec”`.

Under `“What to check?”`, add an ***invariant***: `x = 4`

- An invariant is an expression that must be true in *every state* of execution.

Save the model (`cmd + s`) and click on TLC Model Checker > Run model to check the invariant.

- Check out the “error trace”.
- The error trace shows that the invariant is violated at some point (`x` takes `3` in the second state).

Go back to the `“Model Overview”` tab and try an invariant `x > 2.` Save and rerun the model.

- If you’re creating a new invariant, uncheck the previous one.
- Is the second invariant valid?

Control Flow (2)

Copy and paste the template into your specification (overwrite the existing code):

- String literals are defined between " and ".
- Sequences (tuples) of values come between `<<` and `>>` and can be indexed using `[]` **(starting at 1)**. **Warning:** sequences are 1-indexed.
- `while` and `if` statements work like in programming languages.
- /\ (and) and /= (not equal) are boolean connectives in TLA+ (we're writing a TLA+ expression in PlusCal).
- `either` describes a *non-deterministic* step.
- `skip` is a no-op step.

Assuming a position at row x and column y, enumerate all legal single-step walks in the maze:

- Complete the code sample by replacing `skip` in each branch.
- Save (`cmd + s`) the changes and translate (`cmd + t`) the PlusCal algorithm.

```
------------------------------ MODULE maze ------------------------------
EXTENDS TLC, Integers, Sequences, FiniteSets

(* --algorithm
begin
variables x = 4, y = 1, maze = <<
  <<"#", " ", "#", "#", " ", "$">>,
  <<"#", " ", "#", " ", " ", "#">>,
  <<"#", " ", " ", " ", "#", "#">>,
  <<"*", " ", "#", "#", "#", "#">>
>>;

\* PlusCal algorithm:
begin Start:
while TRUE do
   either \* up
     if x > 1 /\ maze[x - 1][y] /= "#" then
       x := x - 1;
     end if;
   or \* down
       \* TODO

       skip;
   or \* left
       \* TODO

       skip;
   or \* right
       \* TODO
       skip;
   end either;
end while;

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

Walk the Maze

Create a new model and give it a name like “`maze_solver`”.

Add an invariant that would represent a solution to the maze.

- **Hint:** write an invariant that gets violated *only* in a state where `x` and `y` point to a position labeled by `$`.

Save and rerun the model.

- Review the error trace and verify if it walks you through the maze.

Labels (3)

Labels represent steps (state transitions) in the algorithm.

- Adding more labels results in a more *granular* model of the system but it often takes a longer time to verify.
- Check out Hillel Wayne's summary on [using labels].

Use labels to mark the direction of next steps in the maze, as in the code sample.

- Save (`cmd + s`) and retranslate (`cmd + t`).
- Rerun the previous (“`maze_solver`”) model.
- Notice the error trace is decorated with labels.

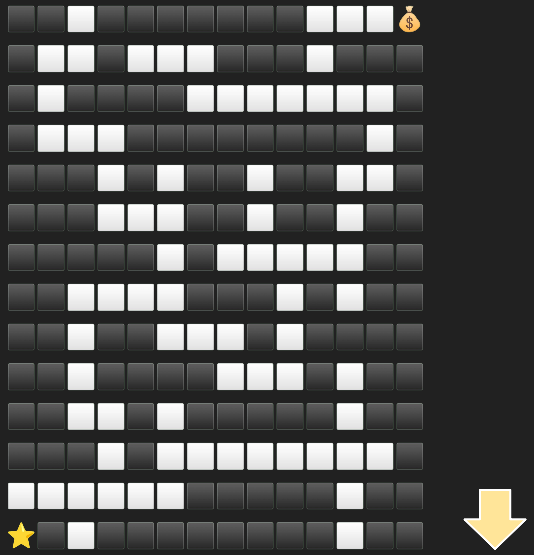
```
------------------------------ MODULE maze ------------------------------
EXTENDS TLC, Integers, Sequences, FiniteSets

(* --algorithm
begin
variables x = 4, y = 1, maze = <<
  <<"#", " ", "#", "#", " ", "$">>,
  <<"#", " ", "#", " ", " ", "#">>,
  <<"#", " ", " ", " ", "#", "#">>,
  <<"*", " ", "#", "#", "#", "#">>
>>;

\* PlusCal algorithm:
begin Start:
while TRUE do
   either
     Up: if x > 1 /\ maze[x - 1][y] /= "#" then
       x := x - 1;
     end if;
   or
     Down: ...
   or
       ...
   or
       ...
   end either;
end while;

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

Maze Grande



(14 x 14)

Breadcrumb (4)

Next, let's try to keep track of the path that takes us to `$`. We'd like to construct a sequence of strings that would show a path through the maze at the final state.

Define a new variable `breadcrumb` and initialize it with an empty sequence.

- `Append(seq, item)` (from `Sequences` module) returns a new sequence obtained by appending `item` to `seq`.
- Use `Append` to extend `breadcrumb` when taking a step.

Save (`cmd + s`) and retranslate (`cmd + t`). Rerun the previous (`"maze_solver"`) model.

- Feel free to stop the run!
- Can you explain what went wrong?

Remove `breadcrumb` and its related changes.

- Save and retranslate the specification.

```
------------------------------ MODULE maze ------------------------------
EXTENDS TLC, Integers, Sequences, FiniteSets

(* --algorithm
begin
variables ..., breadcrumb = <<>>;

\* PlusCal algorithm:
begin Start:
while TRUE do
   either
     Up: ...
       breadcrumb := Append(breadcrumb, "Up");
     end if;
   or
     Down: ...

       \* TODO

   or

     Left: ...

       \* TODO

   or
     Right: ...

       \* TODO

   end either;
end while;

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

Sets

Sets are collections of *unordered* values:

- `{4, 8, 15, 16, 23, 42}` is a set of numbers.
- `1..5` is a syntax sugar for `{1, 2, 3, 4, 5}`.

The expression `x \in s` returns `TRUE` if `x` is a member of set `s` and `FALSE` otherwise.

Filtering: `{x \in s: P(x)}` returns a subset of `s` for which predicate `P` is true.

- `{i \in 1..10: i % 3 = 0}` is the set `{3, 6, 9}`.

Mapping: `{P(x): x \in s}`, returns a set obtained by applying `P` to the elements of set `s`.

- `{Len(i) * 2: i \in {"a", "b", "abc"}}` is the set `{2, 6}`.

`CHOOSE x \in s: P(x)` returns an (arbitrary) element of `s` for which predicate `P` is `TRUE`; the execution **fails** if such an element doesn't exist.

- Although `CHOOSE` returns an arbitrary element of the set, its behavior is deterministic in every run.
- `CHOOSE i \in 1..10: i % 3 = 0` *consistently* returns one of 3, 6, or 9.

TLA+ Variables (5)

Move `maze` to the TLA+ code before the algorithm:

- We can define variables directly in TLA+, outside of `--algorithm` comments.
- Notice that the TLA+ syntax is slightly different (use `==` with no ‘`;`’ at the end).

Use the operator `Len` (from `Sequences`) to define a value `Height` (directly in TLA+). Use `Height` in your PlusCal algorithm (e.g., to initialize `x`, etc.)

The operator `DOMAIN` returns a **set** of valid indices for a given sequence. We will revisit this operator later.

- `DOMAIN <<3, 4, 5>>` is `{1, 2, 3}`.
- Remember sequences are 1-indexed!

Assuming all elements in the maze are of the same length, define a value `Width` and use it in your PlusCal algorithm.

- **Hint:** use `CHOOSE` to pick any index from `DOMAIN` of `maze` and return the length of its corresponding sequence in `maze`.

```
------------------------------ MODULE maze ------------------------------
EXTENDS TLC, Integers, Sequences, FiniteSets

maze == <<
 <<"#", " ", "#", "#", " ", "$">>,
 <<"#", " ", "#", " ", " ", "#">>,
 <<"#", " ", " ", " ", "#", "#">>,
 <<"*", " ", "#", "#", "#", "#">>
>>
Height == Len(maze)
Width == ...
(* --algorithm
begin
variables x = Height, y = 1;

\* PlusCal algorithm:
begin Start:
while TRUE do
   either
     Up: if x > 1 /\ maze[x - 1][y] /= "#" then
       x := x - 1;
     end if;
   or
     Down: if x < Height /\ maze[x + 1][y] /= "#" then
       x := x + 1;
     end if;
    ...
   end either;
end while;

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

Operators

Operators are TLA+ definitions that take arguments. Operators act similar to functions in (functional) programming languages:

- `Double(x) == 2 * x`
- `Max(x, y) == IF x > y THEN x ELSE y` (we are in TLA+ territory!)

Recursive operators must be declared using `RECURSIVE`:

- ```
  RECURSIVE Count(_, _)
  Count(seq, item) == IF seq = <<>>
        THEN 0
        ELSE IF Head(seq) = item
              THEN Count(Tail(seq), item) + 1
              ELSE Count(Tail(seq), item)
  ```

`Head` and `Tail` are operators form module `Sequences`:

- `Head s` returns the first item of sequence `s`.
- `Tail s` returns `s` without its first item.
  ```

Assumptions (6)

Copy and paste the definition of `Count` from the previous page into the TLA+ code before the PlusCal section.

Define an operator `CountAll`, which counts the occurrences of an item in a sequence of sequences:

- You may use `Count` in the definition of `CountAll`.
- **Hint:** Your definition would look very similar to `Count`.

They keyword `ASSUME` makes assertions about *constant* values.

- It's common to use assumptions for input validation and *type invariants*.
- For example, `ASSUME maze /= <<>>` assumes that the input maze isn't empty.

Write two assumptions to ensure that there is exactly one starting point `"*"` and exactly one endpoint `"$"` in `maze`.

```
------------------------------ MODULE maze ------------------------------
...

RECURSIVE Count(_, _)
Count(seq, item) == IF seq = <<>>
   THEN 0
   ELSE IF Head(seq) = item
      THEN Count(Tail(seq), item) + 1
      ELSE Count(Tail(seq), item)

RECURSIVE CountAll(_, _)
CountAll(seqs, item) == ...

ASSUME ...
ASSUME ...


(* --algorithm

begin

...

begin Start:

while TRUE do

  ...

end while;

end algorithm; *)

=========================================================================
```

More Assumptions (7)

`\A x \in s: P(x)` returns `TRUE` if a predicate `P` is true about all elements of a *set* `s`; otherwise, it returns `FALSE`.

- `\A` is in fact a logical *forall* (∀) quantifier.
- `\A x \in {4, 2, 42}: x % 2 = 0` is `TRUE` but `\A x \in 4..10: x > 4` is `FALSE`.
- You may think of `\A` statements as predicates for higher-order functions (e.g., map and filter) in (functional) programming languages.
- Similarly, `\E` is a logical exists (∃) quantifier.

Add an assumption ensure all sequences in `maze` are of the same length (i.e., `Width`).

- You may use the code in the template.

Write an assumption that ensures `maze` consists of only these strings `{"*", "$", "#", " "}` .

- You would need to use nested `\A`s to range over the domain of `maze` as well as the domains of its elements.

```
------------------------------ MODULE maze ------------------------------
...

ASSUME \A i \in DOMAIN maze : Len(maze[i]) = Width
ASSUME ...

(* --algorithm

begin

...

begin Start:

while TRUE do

  ...

end while;

end algorithm; *)

==========================================================================
```

The Starting Position (8)

TLA+ structures resemble structures (records) in programming languages; they map field names to values:

- `s == [a |-> "lol", b |-> {1, 2, 42}]` is a structure with keys `a` and `b`.
- Values are accessed by either `s.a` or `s[a]`.

Use semicolons (`:`) to create sets of structures:

- `[num |-> 42, color : {"Red", "Green", "Blue"}]` returns a set of 3 structures. The value of `num` in all structures is `42` but `color` picks different values.
- `[num : 1..42, color : {"Red", "Green", "Blue"}]` is a set of 126 structures!

Replace `x` and `y` with a structure `pos` with two fields `x`, `y`:

- Write an expression to initialize `pos` with the position of `"*"` in `maze`.
- **Hint:** use `CHOOSE` to pick a position from a set of structures, ranging over `1..Width` and `1..Height`.

```
------------------------------ MODULE maze ------------------------------
...

(* --algorithm
variable pos = CHOOSE ...
...
begin Start:

while TRUE do
   either
     Up: if pos.x > 1 /\ maze[pos.x - 1][pos.y] /= "#" then
       pos.x := pos.x - 1;
     end if;
   or
     Down: if pos.x < Height /\ maze[pos.x + 1][pos.y] /= "#" then
       pos.x := pos.x + 1;
     end if;
   or
     Left: if pos.y > 1 /\ maze[pos.x][pos.y - 1] /= "#" then
       pos.y := pos.y - 1;
     end if;
   or
     Right: if pos.y < Width /\ maze[pos.x][pos.y + 1] /= "#" then
       pos.y := pos.y + 1;
     end if;
   end either;
end while;
*)
=========================================================================
```

Constants (9)

Replace the definition of `maze` with `CONSTANT maze`:

- Now, your specification reads the value of `maze` from its models.
- You may define multiple constants, separated by comma.

Save (`cmd + s`) and translate (`cmd + t`) the changes.

In `“Model Overview”` tab, under `“What is the model?”`, edit the value of `maze`:

- Try *Maze Venti* from the next slide.
- Save and rerun the model.
- You would need to replace `x` and `y` in your invariant with `pos.x` and `pos.y`.

```
------------------------------- MODULE maze -------------------------------
CONSTANTS maze
...

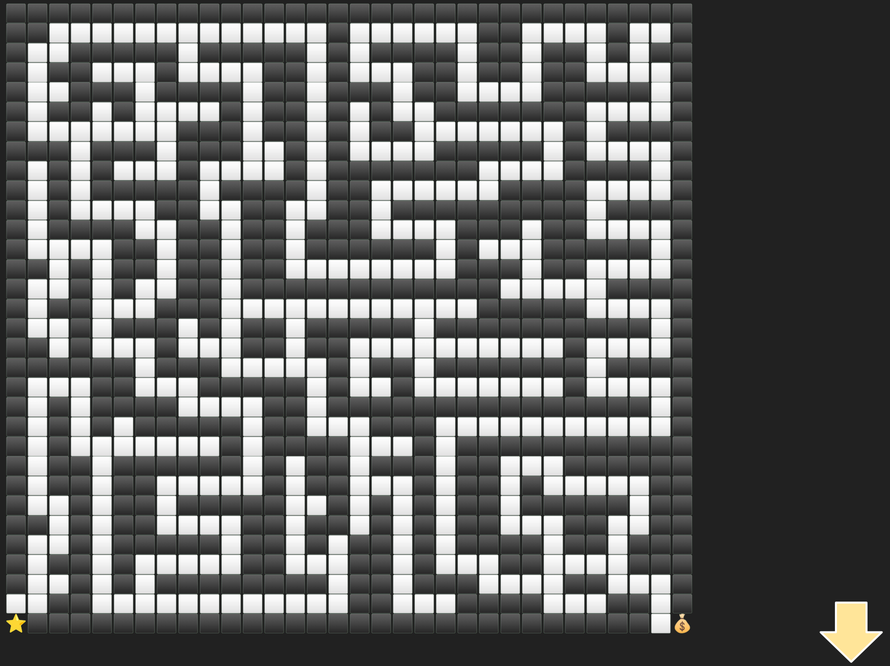

(* --algorithm
begin
variables pos = CHOOSE p \in [x : 1..Height, y : 1..Width] :
    maze[p.x][p.y] = "*"

\* PlusCal algorithm:
begin Start:
while TRUE do
    either
      Up: if pos.x > 1 /\ maze[pos.x - 1][pos.y] /= "#" then
        pos.x := pos.x - 1;
      end if;
    or
      ...
end while;


end algorithm; *)

=============================================================================
```

Maze Venti



Conflict-Free Replicated Data Types (CRDTs)

CRDTs come in two flavors, state-based and operation-based.

- They are used to achieve *eventual consistency* in distributed environments.
- CRDTs allow asynchronous communication and are guaranteed to converge despite message duplication and reordering. No a priori coordination is needed.

State-based CRDTs

- Every member maintains a *local* replica of the CRDT object.
- Members can `update` their local replicas and broadcast them to all other members.
- Upon receipt of *remote* instances, members `merge` them with their local replicas.
- All local replicas converge on the same value as long as all messages are *eventually* delivered.

To make this work, `update` has to be a *monotonic* operation. Also, `merge` must be:

- *idempotent:* i.e., `merge(x, x) = x`
- *associative:* i.e., `merge(x, merge(y, z)) = merge(merge(x, y), z)`
- *commutative:* i.e., `merge(x, y) = merge(y, x)`

Distributed Counter (Take 1)

We’re going to model a grow-only counter in a distributed environment. Every node maintains a local integer, representing the value of a counter.

`update` increments the local replica.

- It’s monotonically increasing (ignoring integer overflow).

`merge` is the maximum (`Max`) of two values.

- `Max(x, x) = x`
- `Max(x, y) = Max(y, x)`
- `Max(Max(x, y), z) = Max(x, Max(y, z))`

We expect all local counters to eventually converge on the same value.

Processes (1)

Create a new specification and call it "`crdt.tla`". Copy and paste the template code into your specification:

- `process Member` defines a process. We create an instance of `Member` for each element in a set `Processes`.
- `local` is a local variable for instances of `Member`.
- **Warning:** If you're planning to use the templates, stick to the name "`crdt`" for your specification!

Any instance of `Member` runs in an infinite loop:

- It *may* (`either`) update and broadcast its `local`.
- Next, it merges any remote value that it receives with its `local`.

Create a new model, namely "`crdt`".

- Under `"What is the model?"`, assign a `"model value"` `NULL` to `NULL`.
- Assign a `"Set model value"` `{p1, p2}` to `Processes`.
- Model values are **nominal**, equal only to themselves.

```
------------------------------ MODULE crdt ------------------------------
EXTENDS Integers, Sequences, FiniteSets

CONSTANTS NULL, Processes

(* --algorithm
begin
process Member \in Processes
  variable local = 0;

  begin Start:
    while TRUE do
      (*
        The process either (non-deterministically) updates (modifies) and
        broadcasts its local value, or it doesn't.
      *)
      either
        Update: \* Update
          local := local + 1;
        Downstream: \* Broadcast
          skip; \* TODO
       or
        skip;
      end either;

      (*
        The process receives a remote value and merges it with its local value.
      *)
      Merge:
        skip; \* TODO
    end while;
  end process;
end algorithm;
*)
==========================================================================
```

Functions

Functions are data types that map keys to values.

- They are similar to maps (dictionaries) in programming languages.
- `func == [i \in S |-> F(i)]` defines a function, namely `func`, over a set `S` as its domain.
- `f == [i \in 1..5 |-> i * i]` is a function from `1` to `1`, `2` to `4`, …, and `5` to `25` (otherwise undefined).
- `g == [s \in {"foo", "", "tla+"} |-> Len(s)]` is a function from `"foo"` to `3`, `""` to `0`, and `"tla+"` to `4`.

We can index into a function to access its values.

- `g["foo"]` returns `3`.

Given a function `f`, `DOMAIN f` returns the *set* over which the `f` is defined:

- `DOMAIN g` returns `{"foo", "", "tla+"}`
- A sequence of length `n` is in fact a function over the domain `1..n` (function `f` is a sequence `<<1, 4, 9, 16, 25>>`).
- That's why when applied to a sequence, `DOMAIN` returns a set of its valid indices.

First Draft (2)

Add the operator `Max` to your specification.

Define a *global* variable `msg_queue` in the PlusCal.

- Initialize it with a function that maps every instance of `Member` to the empty sequence `<<>>`.

Add the code under `Downstream` to your specification.

- `self` is a special variable that holds the unique identifier of the running process.

Complete `Member`'s behavior under `Merge`.

- If the current (`self`) process has received any messages (in `msg_queue`), then:
- Assign one (e.g., `Head`) of the received messages to a new local variable, namely `remote`.
- Remove the message from the queue of the current process (consider using `Tail`).
- Update `local` by merging its current value and remote using `Max`.

```
------------------------------ MODULE crdt ------------------------------
CONSTANTS NULL, Processes

Max(x, y) == IF x > y THEN x ELSE y

(* --algorithm
begin
 variable msg_queue = ...

process Member \in Processes
 variable local = 0, remote = NULL;;

 begin Start:
  while TRUE do
    ...
    Downstream: \* Broadcast
     msg_queue := [
      q \in DOMAIN msg_queue |-> IF q = self
       THEN msg_queue[q]
       ELSE Append(msg_queue[q], local)
     ];
    ...
   Merge:
    skip; \* TODO

  end while;
 end process;
end algorithm;
*)
=========================================================================
```

Safety Properties

Save and retranslate the specification. Create a new model, namely `“crdt_model”`.

Add an invariant under `“What to check?”`: `local[p1] = local[p2]`

- Recall that TLC only understands the TLA+ translation.
- Local values of PlusCal processes are translated as functions, with processes as keys.
- This invariant fails. Check out the error trace.

Invariants are used to express **safety** properties; ensuring that something *bad* won’t happen.

Liveness Properties

Liveness properties assert that something *good* will eventually happen.

TLA+ offers a rich set of operators to express temporal properties that span across many states:

- `[]P` (box operator) means statement `P` is always true.
- `<>P` (diamond operator) means `P` will *eventually* become true.
- `<>[]P` means `P` will *eventually always* become true.

The property of our interest is: `<>[](local[p1] = local[p2])`

- Local values will eventually become and remain equal (i.e., eventual consistency).
- Add this expression as a new **property** under `“What to check?”`.
- Uncheck other invariants/properties.
- Save (`cmd + s`) and run the model.

Fairness (3)

The previous property fails because of *stuttering steps* (see the last line of the error trace).

- Stuttering steps happen when a system can but doesn't take a step. They are useful when modeling real systems (see here).
- Sometimes, we may think of stuttering as a system crash, where the system refuses to continue.

A *fair* system eventually takes any available steps.

- Fairness usually goes hand-in-hand with liveness.
- Use the keyword `fair` in PlusCal to make `Member` a fair process.
- Analyzing stuttering systems is out of the scope of this workshop.

Make `Member` a `fair` process.

- Save and retranslate your specification.
- Rerun the model on previous property. Does the new behavior look familiar?

```
------------------------------ MODULE crdt ------------------------------
CONSTANTS NULL, Processes

Max(x, y) == IF x > y THEN x ELSE y

(* --algorithm
begin
  variable msg_queue = [p \in Processes |-> <<>>];


fair process Member \in Processes
  variable local = 0, remote = NULL;

  begin Start:
    while TRUE do
      ...
      Downstream: \* Broadcast
        msg_queue := [
          q \in DOMAIN msg_queue |-> IF q = self
            THEN msg_queue[q]
            ELSE Append(msg_queue[q], local)
        ];
      ...
     Merge:
      ...
    end while;
  end process;
end algorithm;
*)
=======================================================
```

Bounding the State Space (4)

Infinite loops are useful for modeling but they may result in an infinite state space. To restrict the state space:

- we can provide TLC with additional parameters to bound the depth of search. This works only for testing safety properties.
- we can restrict the specification. One idea is to allow only a limited number of updates and broadcasts.

Small Scope Hypothesis:

"... if the analysis [exhaustively] considers all small instances, most flaws will be revealed."

-- *Daniel Jackson*

Define a new `CONSTANT`, namely `MaxUpdates`:

- Modify the PlusCal algorithm to allow at most `MaxUpdates` updates and broadcasts per process instance.
- Save and retranslate the specification.
- Under `"What is the model?"`, assign 5 to `MaxUpdates`. Save and run the model.

```
------------------------------ MODULE crdt ------------------------------
CONSTANTS NULL, Processes, MaxUpdates

...

(* --algorithm
begin
...

fair process Member \in Processes
variable local = 0, remote = NULL, updates = 0;
 ...
    if updates < MaxUpdates then
      updates := updates + 1;
      either
        Update: \* Update
          local := local + 1;
        Downstream: \* Broadcast
          msg_queue := [
              q \in DOMAIN msg_queue |-> IF q = self
              THEN msg_queue[q]
              ELSE Append(msg_queue[q], local)
          ];
      or
        skip;
      end either;
    end if;
...
end algorithm;
*)
=========================================================================
```

Oracle (5)

Local values of the processes converge on the same value, but is that our expect evaue, i.e., the sum of increments?

Define a *global* PlusCal variable `oracle`, initialize it with 0, and increment it every time a process increments its `local`.

- Save and translate the changes.

Add a new property to check if `local` of `p1` and `p2` are *eventually* equal to `oracle`.

- Use the `<>[]` operator; also, recall `/\` is logical “and” in TLA+.
- Save and run the model.
- **Hint:** You can change `MaxUpdates` to 1 to get a shorter error trace.
- What are you learning from the trace?

```
------------------------------ MODULE crdt ------------------------------
CONSTANTS NULL, Processes, MaxUpdates

...

(* --algorithm
begin
  variable msg_queue = ..., oracle = 0;

fair process Member \in Processes
variable local = 0, remote = NULL, updates = 0;
  ...
      if updates < MaxUpdates then
        updates := updates + 1;
        either
          Update: \* Update
            local := local + 1;
            oracle := oracle + 1;
          Downstream: \* Broadcast
            msg_queue := [
                q \in DOMAIN msg_queue |-> IF q = self
                THEN msg_queue[q]
                ELSE Append(msg_queue[q], local)
            ];ß
        or
          skip;
        end either;
      end if;
...
end algorithm;
*)
=====================================================
```

Grow-Only Counter (G-Counter)

G-Counters are well-known CRDTs that implement increment-only counters in distributed environments. We are going to modify our specification to model a G-Counter and verify its properties.

The CRDT object for a G-Counter is a vector of integers, where each index of the vector is associated with a member:

- To increment the counter (`update`), each process increments the integer at its own index.
- To `merge` two replicas, take the maximum of the two integers at every index of the vector.
- The *value* of a vector is computed by summing all of its elements.

G-Counter (6)

Use a function to represent a G-Counter vector and assign it to `local`.

- Write an expression, like the one for `msg_queue`, to initialize `local` with 0s.
- Change the code under `Update` and `Merge` as needed to work with the new definition of `local`.

Add the new operators at the top of the code template to your specification:

- We're going to use `Sum` in our invariant to compute the value of G-Counters.
- **Helpers:** `Fold` is the conventional fold (reduce) function for sequences. `FuncValues` returns the values of a function in a sequence.

Modify the existing properties in your model to apply `Sum` to local values.

- Save and rerun the model.

```
------------------------------- MODULE crdt -------------------------------
...
RECURSIVE Fold(_, _, _)
Fold(Op(_, _), s, r) == IF s = <<>>
  THEN r
  ELSE LET x == Head(s)
    IN Fold(Op, Tail(s), Op(x, r))

RECURSIVE FuncValuesHelper(_, _)
FuncValuesHelper(f, d) == IF d = {}
  THEN <<>>
  ELSE LET k == CHOOSE i \in d: TRUE
    IN Append(FuncValuesHelper(f, d \ {k}), f[k])

FuncValues(f) == FuncValuesHelper(f, DOMAIN f)

Sum(f) ==
  LET op(x, y) == x + y
      vs == FuncValues(f)
  IN Fold(op, vs, 0)

(* --algorithm
begin
  variable msg_queue = [ p \in Processes |-> <<>> ], oracle = 0;

fair process Member \in Processes
variable local = ..., remote = NULL, updates = 0;
 ...
        Update: \* Update
          local[self] := ...
          oracle := oracle + 1;
          ...
     Merge:
          ...
          local := ...
...
*)
```

Modeling the Network (7)

It's common to specify the network's behavior as a process that interferes with the communicating messages; it may arbitrarily duplicate, reorder, or remove some messages.

Copy and paste the definition of `Network` into your PlusCal code. Add `Drop`, `Reorder` and `MaxDups` as `CONSTANTS`.

- `Drop` and `Reorder` takes a boolean value (`TRUE` or `FALSE`) to allow message loss and reordering.
- `MaxDups` limits the number of duplicate messages to avoid non-termination.
- Save and retranslate the specification.

In your model, assign `FALSE` to `Drop`, `TRUE` to `Reorder`, 3 to `MaxDups` and 3 to `MaxUpdates`.

- Save and rerun the model.
- **Warning:** assigning greater values to `MaxDups` and `MaxUpdates` would dramatically increase the execution time.
- *The run takes up to 1 minute to complete!*

```
------------------------------ MODULE crdt ------------------------------
CONSTANTS ..., Drop, Reorder, MaxDups
...

(* --algorithm
...
process Network = 0

  variable i = 1, dups = 0,

    proc_ids = CHOOSE p \in [ 1..Cardinality(Processes) -> Processes] : TRUE;

  begin NetStart:
    while TRUE do
      if msg_queue[proc_ids[i]] /= <<>> then
        either \* Drop a message
          if Drop then
            msg_queue[proc_ids[i]] := Tail(msg_queue[proc_ids[i]]);
          end if;
        or \* Duplicate a message
          if dups < MaxDups then
            msg_queue[proc_ids[i]] :=
              Append(msg_queue[proc_ids[i]], Head(msg_queue[proc_ids[i]]));
            dups := dups + 1;
          end if;
        or \* Reorder a message
          if Reorder then
            msg_queue[proc_ids[i]] := Append(
              Tail(msg_queue[proc_ids[i]]),
              Head(msg_queue[proc_ids[i]])
            );
          end if;
        end either;
      end if;
      i := (i % Cardinality(Processes)) + 1;
    end while;
  end process;
...
*)
```

More Analysis

Lossy network

- Verify that in a lossy network (`Drop = TRUE`), CRDTs may not converge.

Multiple processes

- Replace the existing property with: `<>[] (\A p \in Processes: Sum(local[p]) = oracle)`
- Check the property for 3 processes (i.e., `Processes = {p1, p2, p3}`)
- **Suggestion:** `MaxUpdates = 1, MaxDups = 5, Drop = FALSE, Reorder = TRUE.`

More processes

- Try `Processes = {p1, p2, p3, p4}`
- **Suggestion:** `MaxUpdates = 1, MaxDups = 0, Drop = FALSE, Reorder = FALSE.`
- **Warning:** greater values for these parameters can cause very long runs.