Model Checking in Practice: Spin and TLA$^+$

November 17, 2026

In the previous exercise sheet, we defined LTSs, LTL and CTL, and we model checked small systems by hand. Here, we use model checking tools that process larger numbers of states.

Part 1: Spin

1 Installing Spin

Spin is a single executable (`spin`). It does *not* verify a model by itself: it translates the model into a C program, `pan.c`, which you then compile and run. So you need a C compiler as well. Recall that, in the snippets below (in grey boxes), lines starting with `$` are commands that you should type and the rest is expected output.

Exercise 1 (Install Spin). Install Spin with the instructions for your platform below.

macOS. Install the command-line developer tools (to access `clang`, `make` and the C preprocessor that Spin calls), then install Spin itself with Homebrew:

```
$ xcode-select --install       # skip if already installed
$ brew install spin
```

Linux (Debian, Ubuntu).

```
$ sudo apt update
$ sudo apt install spin build-essential
```

Windows. The simplest route is the Windows Subsystem for Linux: run `wsl --install` in PowerShell, then follow the Linux instructions inside the Ubuntu shell. Precompiled Windows binaries are also available from https://spinroot.com/spin/Src/.

From source (any platform).
If the package is missing or too old, build it with `gcc` (or `clang`), `make` and `yacc`/`bison`:

```
$ git clone https://github.com/nimble-code/Spin.git
$ cd Spin/Src
$ make
$ sudo make install
```

Note: Regardless of your installation path, check that Spin runs:

```
$ spin -V
Spin Version 6.5.2 -- 6 December 2019
```

Exercise 2 (The three-stage workflow). Spin is not one program but a pipeline of three steps. It starts with a Promela (PROcess MEta LAnguage) program (a *model*). Let us start with a simple Promela model: one process that prints a line and stops. Save it as `hello.pml`:

```
active proctype hello() {
    printf("hello from Promela\n")
}
```

Part 1 — *Running the stages.* Now run the four commands below one at a time, and answer the question next to each. Only one of the four commands tells you whether the model is correct.

```
$ spin hello.pml              # what was the printed output?
$ spin -a hello.pml           # what files were created?
$ gcc -o pan pan.c            # what file was created?
$ ./pan                       # how many errors does it report?
```

Part 2 — *Changing the model.* The statement `assert(false)` fails whenever it is reached, so the verifier should report it as an error. Add it at the end of the process. Promela separates statements with a semicolon, so the `printf` line now needs one:

```
active proctype hello() {
    printf("hello from Promela\n");
    assert(false)
}
```

Which of the previous commands must you repeat before `./pan` reports the failing assertion?

2 Writing a transition system in Promela

Let us write the vending machine (Figure 1) of the LTL exercise sheet in Promela. A Promela model is built from *processes* which require specifying:

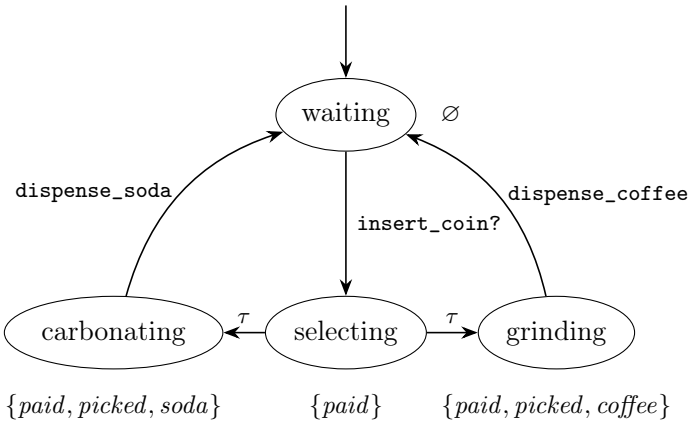


Figure 1: The beverage vending machine.

- The declaration `mtype = { ... }` that introduces symbolic constants (that can stored in variables and compared). In the example below, we write one per state.
- The declaration `active proctype` *name*`() { ... }` that introduces a process that launches when the model starts.

- The block `do :: guard -> statement ... od`, with a set of *guarded commands* or *options*, executed as a loop.[1] The *guard* is the boolean expression between `::` and `->`. On each pass, Spin picks *any one* option whose guard is true and runs the statement.

So the skeleton of the vending machine is:

```
mtype = { waiting, selecting, _____, _____ };

mtype state = waiting;

active proctype machine() {
    do
    :: state == waiting     -> state = selecting     /* insert_coin? */
    ...
    od
}
```

Each option of the loop is one arrow of Figure 1: the guard says in which state the model must be to transition, and the statement moves us to the next one.

Exercise 3 (Writing the transitions). Write in a script the skeleton and the remaining options in the body of the `do`-loop that implements the remaining transitions of Figure 1. Save the file as `vending.pml`. The skeleton above already shows *waiting* $\longrightarrow$ *selecting*. Then, check that Spin accepts it with `spin -a vending.pml` (it does not check that you added all options). The list of missing transitions is below.

Hint. The guard and the statement are separated by `->`, and options are separated by `::`. Assignment is `=` and comparison is `==`, as in C.

a) *selecting* $\longrightarrow$ *carbonating*

b) *selecting* $\longrightarrow$ *grinding*

c) *carbonating* $\longrightarrow$ *waiting*

d) *grinding* $\longrightarrow$ *waiting*

Exercise 4 (Simulations). Let us interact more with spin to understand its behaviour.

Part 1 — *Several options at once.* Run the following command from two to five times:

```
$ spin -u12 -p vending.pml
```

Compare the traces and answer. Are all of them equal? From what you observe, is a Spin simulation deterministic?

Hint. The `-u12` flag stops each run after at most 12 steps, and `-p` prints each executed statement.

Part 2 — *No option at all.* Now delete the option for *grinding* $\longrightarrow$ *waiting* from `vending.pml` and run the pipeline steps:

```
$ spin -a vending.pml
$ gcc -o pan pan.c
$ ./pan
```

[1] The Spin manual calls them *option sequences*: https://spinroot.com/spin/Man/do.html.

What does `pan` report? Simulate again (`spin -u12 -p vending.pml`): various times. Do all runs reach the depth of 12? Next, restore the transition for *grinding* ⟶ *waiting*, but replace the right-hand-side of the arrow (`->`) with `break`:

```
:: state == grinding -> break
```

Regenerate `pan.c`, compile it, and run `./pan` again. Does the report change? What is different about a run that reaches `grinding` now?

3 Writing properties in Promela

Figure 1 labels each state with the atomic propositions that hold in it. Promela has no labels. Instead, an atomic proposition is a *boolean expression over the variables* of the model, given a name with `#define`. For instance, *paid* holds in every state except *waiting*:

```
#define paid  (state != waiting)
```

A name defined this way is a *macro*: "a fragment of code which has been given a name. Whenever the name is used, it is replaced by the contents of the macro."[2] Spin then evaluates the expression in each state it visits.

Exercise 5 (Naming propositions). Add the macro `paid` to `vending.pml`, below the declaration of `state`. Then, add one macro for each proposition listed below to match the labels in Figure 1. For the last one, *idle*, define it as true exactly when the machine is in `waiting`. We will use it in the next section.

Hint. Promela uses C's boolean operators: `==`, `!=`, `||`, `&&` and `!`. Write the defining expressions in parentheses, as it was done for `paid`.

a) *picked*

b) *soda*

c) *coffee*

d) *idle*

To write LTL formulas (or *claims*) in Promela, we use the keyword `ltl`, a name, and an LTL formula in braces. The atomic propositions should be the macros. For instance, the claim *every paid drink is eventually picked* is represented via:

```
ltl served { [] (paid -> <> picked) }
```

The formula is spelled in ASCII, since LTL's symbols cannot be typed in a plain text file.

Exercise 6 (Promela's LTL operators). Match each LTL connective to its spelling in Promela by writing its letter in the brackets.

Hint. The claim `served` above uses three of the operators. The Spin manual lists all of them: https://spinroot.com/spin/Man/ltl.html.

[2] The GNU C preprocessor manual, https://gcc.gnu.org/onlinedocs/cpp/Macros.html. Spin passes every model through the C preprocessor before reading it.

LTL	**Promela**
a) $\mathrm{G}\,\varphi$	( ) `phi -> psi`
b) $\mathrm{F}\,\varphi$	( ) `<> phi`
c) $\mathrm{X}\,\varphi$	( ) `phi W psi`
d) $\varphi\,\mathrm{U}\,\psi$	( ) `!phi`
e) $\varphi\,\mathrm{W}\,\psi$	( ) `X phi`
f) $\neg\varphi$	( ) `phi <-> psi`
g) $\varphi \wedge \psi$	( ) `[] phi`
h) $\varphi \vee \psi$	( ) `phi || psi`
i) $\varphi \rightarrow \psi$	( ) `phi U psi`
j) $\varphi \leftrightarrow \psi$	( ) `phi && psi`

Exercise 7 (Writing claims).

Part 1 — *Writing them.* Add the claim `served` to the end of `vending.pml`. Then add one claim for each property below, with the name given, using the macros of Exercise 5.

a) `onedrink`: the machine is never making coffee and soda at the same time.

b) `returns`: the machine is waiting for a coin infinitely often.

c) `fairsoda`: sodas are made infinitely often.

4 Reading Spin's Outputs

Exercise 8 (Running the verifier). Running `spin -a vending.pml` will throw you a warning about the model containing several claims. Therefore, after compiling, use `pan` as suggested by the tool to tell which one to check, with the `-N` flag. The `-a` flag (for `./pan`) turns on the search for the infinite behaviours that properties with the shape `[]<>` need. Which of the properties we have written does not hold for the vending machine? Confirm that with the tool (recording the number of errors). Once an error is reported, you will see a new file. What is its extension?

```
$ spin -a vending.pml
$ gcc -o pan pan.c
$ ./pan -a -N served
$ ./pan -a -N onedrink
$ ./pan -a -N returns
$ ./pan -a -N fairsoda
```

Exercise 9 (Reading the verifier's report). Each item below represents a comment explaining a line of the report of `./pan -a -N fairsoda`. Write the letter of each comment after the `//` of the line it explains. An example is provided.

a) The four lines below this one list the kinds of error the verifier can look for: `+` means that it looked for them, and `-` that it did not.

a1) Because of the `-a` flag, the verifier looked for runs that end in a loop, which Spin calls *acceptance cycles*.

a2) While a claim is checked, the verifier does not look for blocked processes (invalid end states), the error of Exercise 4.

a3) Spin turned the claim chosen with `-N` into a *never claim*: a description of the runs that must never happen, that is, of the negation of the property.

a4) The verifier also looked for failing `assert` statements, as in Exercise 2. This model has none.

b) The verifier found a run that violates the claim and ends in a loop that repeats for ever, which Spin calls an *acceptance cycle*. The loop begins at depth 2 of the run.

c) One violation was found, and the search went at most 13 steps deep.

d) Warning: The verifier stopped at the first violation, so it did not explore every state.

e) The counterexample was saved to a `.trail` file, so that it can be replayed.

f) An (partial order) optimisation that lets the verifier explore fewer states is activated.

```
pan: ltl formula fairsoda
pan:1: acceptance cycle (at depth 2)                          // ____
pan: wrote vending.pml.trail                                  // (e)

(Spin Version 6.5.2 -- 6 December 2019)
Warning: Search not completed                                 // ____
        + Partial Order Reduction                             // ____

Full statespace search for:                                   // ____
        never claim             + (fairsoda)                  // ____
        assertion violations    + (if within scope of claim)  // ____
        acceptance   cycles     + (fairness disabled)         // ____
        invalid end states      - (disabled by never claim)   // ____

State-vector 28 byte, depth reached 13, errors: 1             // ____
        9 states, stored (11 visited)
        0 states, matched
       11 transitions (= visited+matched)
        0 atomic steps
hash conflicts:         0 (resolved)
...
```

Exercise 10 (Reading a counterexample). We can replay the `.trail` file with Spin as below, where the `-p` flag is for printing each step:

```
$ spin -t -p vending.pml
```

Each item below represents a comment explaining a line of the replay. Write the letter of each comment after the `//` of the line it explains. An example is provided.

a) The replay starts listing the number of the claim that failed: `fairsoda`.

b) The machine executes the guard of the option for *waiting* $\longrightarrow$ *selecting*.

c) The loop that repeats for ever begins here, at the depth given by `pan:1: acceptance cycle (at depth 2)` in Exercise 9.

d) The second step of the same option. In Promela, the guard `state == waiting` is a statement of its own, and `->` separates two statements exactly as `;` does.[3] So an option runs in two steps: first the comparison `==`, which only waits until it is true and changes nothing, then the assignment `=`, which changes the state to `selecting`.

e) At step 6 the machine took the option at line 15, for *selecting* ⟶ *grinding* (as opposed to *selecting* ⟶ *carbonating*).

f) The machine is back in `waiting`, where the loop began, so the loop can repeat for ever.

g) The replay has reached the end of the trail file after 14 steps.

h) The value of each variable when the replay ends.

After printing the claims again, the replay shows:

```
...
starting claim 4                                                    // _____
Never claim moves to line 34    [(!((state==carbonating)))]
  2:    proc  0 (machine:1) vending.pml:13 (state 1)    [((state==waiting))]   // (b)
  <<<<<START OF CYCLE>>>>>                                          // _____
Never claim moves to line 39    [(!((state==carbonating)))]
  4:    proc  0 (machine:1) vending.pml:13 (state 2)    [state = selecting]    // _____
  6:    proc  0 (machine:1) vending.pml:15 (state 5)    [((state==selecting))] // _____
  8:    proc  0 (machine:1) vending.pml:15 (state 6)    [state = grinding]
 10:    proc  0 (machine:1) vending.pml:17 (state 9)    [((state==grinding))]
 12:    proc  0 (machine:1) vending.pml:17 (state 10)   [state = waiting]
 14:    proc  0 (machine:1) vending.pml:13 (state 1)    [((state==waiting))]   // _____
spin: trail ends after 14 steps                                     // _____
#processes: 1
                state = waiting                                     // _____
 14:    proc  0 (machine:1) vending.pml:13 (state 2)
 14:    proc  - (fairsoda:1) _spin_nvr.tmp:38 (state 10)
1 processes created
```

Part 2: TLA$^+$

5 Installing and running TLA$^+$

The TLA$^+$ tools are written in Java. We will use them through the official Visual Studio Code extension, which carries them inside itself, including the model checker TLC. TLC generates the behaviours that a specification allows and checks them against the properties we ask it to check [4].

Exercise 11 (Installing TLA$^+$). Install the three requirements below, in order.

A Java runtime, version 11 or newer. Check what you already have:

```
$ java -version
openjdk version "25.0.2" 2026-01-20
```

[3] The Spin manual: https://spinroot.com/spin/Man/separators.html.

If the command is not found, or reports a version older than 11, install a current JDK, for instance Eclipse Temurin, or `brew install openjdk` on macOS, or `sudo apt install default-jdk` on Debian and Ubuntu.

Visual Studio Code from `https://code.visualstudio.com`.

TLA$^+$ extension. In VSCode, via the Extensions view (`Ctrl+Shift+X`, or `Cmd+Shift+X` on macOS), search for *TLA+*, and install the official one by `tlaplus` (id `tlaplus.vscode-ide`).

Check. Open VSCode's Command Palette (`F1`, or `Cmd+Shift+P` on macOS), type *TLA+*, and find `TLA+: Check model with TLC`. You will run this command with a `.tla` file active in the next exercise. The extension's Getting Started guide shows the model-checking view.

Exercise 12 (The TLC workflow). A TLA$^+$ model consists of two files: a *specification* (`.tla`), describing the system, and a *configuration* (`.cfg`), indicating TLC what to check. Let us start with a system whose single variable counts $0, 1, 2, 0, 1, 2, \ldots$ Save it as `Hello.tla`:

```
---- MODULE Hello ----
EXTENDS Naturals
VARIABLE x
Init == x = 0
Next == x' = (x + 1) % 3
Spec == Init /\ [][Next]_x
Small == x < 3
====
```

and save the configuration, in the same directory, as `Hello.cfg`:

```
SPECIFICATION Spec
INVARIANT Small
```

Part 1 — *Running TLC.* In VSCode, open the folder containing both files, then select the `Hello.tla` tab. Save both files. Open the Command Palette (`F1`, or `Cmd+Shift+P` on macOS), choose `TLA+: Check model with TLC`, and wait for the *TLA+ model checking* tab. The screenshot below shows the result you should see (the times may differ). Match each description to the numbered part of the interface by writing its letter in the corresponding brackets below the screenshot. An example is provided.

a) This button runs the same model check again after you save your changes.

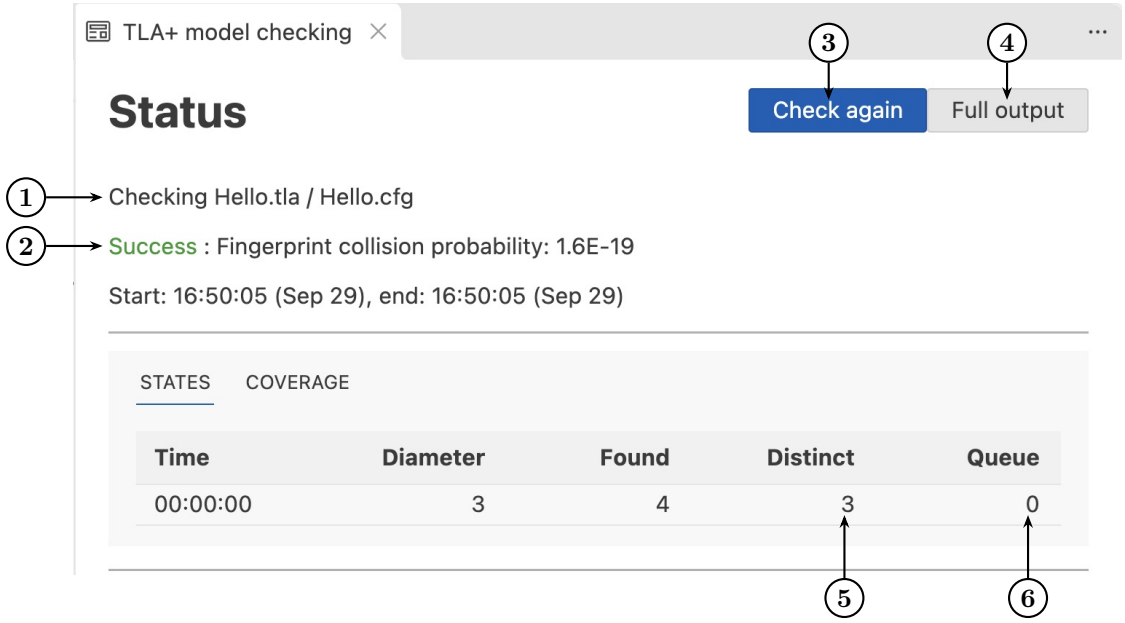
b) This value displays the count of the different states reached: here, three.

c) The filenames identify the specification and configuration used for this run.

d) This status says that TLC completed the check without finding a violation of `Small`.

e) This button opens TLC's full text report.

f) This value counts states still waiting to be explored: here, none.



1 (c) **2** ( ) **3** ( ) **4** ( ) **5** ( ) **6** ( )

Part 2 — ***Breaking the invariant.*** Change the line in `Hello.tla` to `Small == x < 2`, save the file, and choose *Check again* in the model-checking tab. Find the invariant violation in the result, then read the states of its counterexample in order. The output lists the values of x. Which of those values first makes `Small` false?
Note: Restore `Small == x < 3` and save `Hello.tla` before continuing.

Exercise 13 (From an LTS to TLA$^+$). Recall that a *labelled transition system* (LTS) is a structure $\langle S, \longrightarrow, L \rangle$, where S is a finite set of states, $\longrightarrow \subseteq S \times S$ is the transition relation, and $L \colon S \to \mathcal{P}(P)$ is the labelling function, which says which atomic propositions of P hold in each state. Sometimes, people also specify states as initial, drawn with an incoming arrow. Notice that a TLA$^+$ specification describes such an LTS.

Part 1 — *Matching.* Match each part of `Hello.tla` and `Hello.cfg` to its meaning by writing its letter in the brackets. An example is provided.

In `Hello.tla` and `Hello.cfg`	**Meaning**
a) the declaration `EXTENDS Naturals`	(c) declares the initial states
b) the declaration `VARIABLE x`	( ) specifies the set S of states: a state gives a value to every declared variable
c) the formula `Init`	( ) represents one state $s \in S$
d) the formula `Next`	( ) specifies the transition relation $\longrightarrow$
e) the step from `x = 1` to `x' = 2`	( ) represents one transition $s \longrightarrow s'$
f) the equation `x = 1`	( ) declares an atomic proposition that holds in some states and not in others
g) the formula `Small == x < 3`	( ) is not part of the LTS: it imports the standard module that defines the natural numbers and the operators `+`, `%` and `<` [4]
h) the line `SPECIFICATION Spec`	( ) asks TLC to check that an atomic proposition p holds in every reachable state, that is, the LTL formula $\Box p$
i) the line `INVARIANT Small`	( ) tells TLC which formula describes the whole LTS: its initial states and its transitions

Part 2 — *Primed variables.* Complete the sentence: In `Next`, the unprimed `x` is the value of `x` in the _____ state, and the primed `x'` is its value in the _____ state.

Note: An *action* is a Boolean formula about the current and the next states. A step satisfies an action when both of these values make the formula true [4]. For example, the action `Next` in `Hello.tla` describes the allowed transitions between states.

6 TLA$^+$ notation

Exercise 14 (Symbols in ASCII).

Part 1 — *Matching.* Every mathematical symbol of TLA$^+$ has an ASCII representation [4]. Match each ASCII representation to the symbol it stands for by writing its letter in the brackets. An example is provided.

ASCII		**Symbol**	
a) `==`	g) `<>`	(a) $\triangleq$, "is defined as"	( ) \, set difference
b) `/\`	h) `\in`	( ) $\in$	( ) $\Rightarrow$
c) `\/`	i) `\cup`	( ) $\vee$	( ) $\Box$
d) `~`	j) `\`	( ) $\Diamond$	( ) $\forall i \in S.\ P$
e) `=>`	k) `\A i \in S : P`	( ) $\exists i \in S.\ P$	( ) $\neg$
f) `[]`	l) `\E i \in S : P`	( ) $\wedge$	( ) $\cup$

Part 2 — *Evaluating expressions.* TLC can evaluate an expression for you. Select the `Hello.tla` tab, open the Command Palette (`F1`, or `Cmd+Shift+P` on macOS), choose `TLA+: Evaluate expression...`, type the expression and press `Enter`. The value appears in the *Output* panel. The command evaluates the expression in the context of the file whose editor is active, so click inside the `Hello.tla` editor again before each evaluation. Predict the value of each expression below, then check it. An example is provided.

Expression	Value
`{1, 2} \cup {2, 3}`	`{1, 2, 3}`
`{1, 2, 3} \ {2}`	_____
`~(1 = 2)`	_____
`\A i \in {1, 2} : i > 1`	_____
`\E i \in {1, 2} : i > 1`	_____

Exercise 15 (Bulleted lists). A long conjunction or disjunction is written as a *bulleted list*: each item begins with `/\` or `\/` and lies completely to the right of it, and the bullets of one list line up precisely [4]. Write each formula below with $\wedge$, $\vee$ and parentheses. An example is provided.

a)

```
/\ p
/\ \/ q
   \/ /\ r
      /\ s
```

$p \wedge (q \vee (r \wedge s))$

b)

```
\/ /\ p
   /\ q
\/ r
```

_____

c)

```
/\ p
/\ \/ q
   \/ r
```

_____

d)

```
\/ p
\/ /\ q
   /\ r
```

_____

e)

```
/\ \/ p
   \/ q
/\ \/ r
   \/ s
```

_____

Exercise 16 (Functions and tuples). A *function* `f` assigns a value `f[c]` to every element `c` of its *domain*. TLA$^+$ writes the argument in square brackets [4]. We will use:

- the notation `[i \in S |-> e]`, i.e. the function with domain `S` mapping every `i` to `e`.
- the notation `[f EXCEPT ![c] = e]`, i.e. the function `f`, except that it maps `c` to `e`.
- the notation `<<x, y>>` represents a tuple with first element `x`, and second element `y`
- the notation `UNCHANGED e` abbreviates `e' = e`, i.e. it says that the expression `e` keeps its value in the next state [4].
- the notation `[Next]_x` represents `Next \/ UNCHANGED x`, i.e. every step is either a step of the action `Next` or a *stuttering* step, one that leaves `x` unchanged [4].

Predict the value of each expression below, then check it with `TLA+: Evaluate expression...`.

Expression	Value
`[i \in {1, 2} |-> 10 * i][2]`	
`[[i \in {1, 2} |-> "n"] EXCEPT ![2] = "t"][2]`	
`[[i \in {1, 2} |-> "n"] EXCEPT ![2] = "t"][1]`	
`<<"fst", "snd">>[1]`	
`<<"fst", "snd">>[2]`	
`[n \in Nat |-> n + 1][41]`	
`[i \in {1, 2} |-> 10 * i]`	
`[i \in {1, 2} |-> "n"]`	
`[[i \in {1, 2} |-> "n"] EXCEPT ![2] = "t"]`	

7 Mutual exclusion

When two processes share a resource, such as a file, each of them has a part of its code that uses the resource: its *critical section* [3]. If both processes are in their critical sections at the same time, they can corrupt the resource. For instance, two processes appending to the same file can interleave their lines. A *mutual-exclusion protocol* (MEP) decides when each process may enter its critical section. Huth and Ryan list four properties that such a protocol for two processes should have [3]:

- *Safety:* "Only one process is in its critical section at any time."
- *Liveness:* "Whenever any process requests to enter its critical section, it will eventually be permitted to do so."
- *Non-blocking:* "A process can always request to enter its critical section."
- *No strict sequencing:* "Processes need not enter their critical section in strict sequence."

In this section we specify their protocol [3] in TLA$^+$ and check the four properties with TLC. Each process `i` is in one of three states: non-critical (`"n"`), trying to enter (`"t"`) or critical (`"c"`). It moves between them by three actions (Figure 2):

- The action `Try(i)` lets a non-critical process request to enter.
- The action `Enter(i)` lets a trying process enter, but only if the other process is non-critical, or if both are trying and it is the turn of `i`. A shared variable `turn` holds the process whose turn it is. On entering, a process gives the turn to the other process.
- The action `Leave(i)` lets a critical process leave.

Exercise 17 (Writing the specification). Save the provided skeleton below as the file `MutualExclusion.tla`. The set `Procs` contains the two processes of the MEP. The variable `st` maps each process to its state (those in Figure 2), so `st[i]` is the state of process i. The definition `Other(i)` maps to the other process, and `turn` indicates whose turn it is. Complete the skeleton by translating each statement into a TLA$^+$ formula. An example is provided.

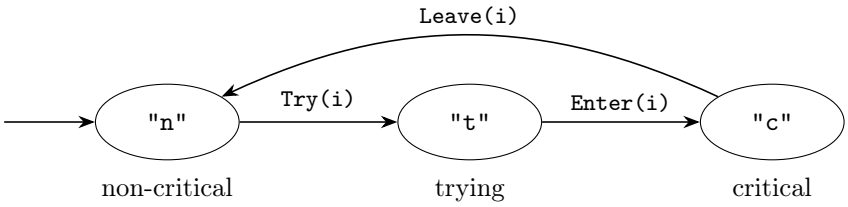


Figure 2: The LTS of a process `i` of the mutual-exclusion protocol.

Note: TLC computes next states from the actions, and every variable must get a complete next value. You can only provide that value from a conjunct of one of the two forms: `x' = e` (assign `x'` the value of `e`) or `x' \in S` (tries every element of the set `S` until `x'` has a value). Other variations produce errors [4]. In particular, when `x` is a function, you cannot write `x'[i] = k`. You must update the whole function, e.g. `x' = [x EXCEPT ![i] = k]`.

a) `Try(i)`: process `i` is non-critical, and in the next state, `i` is trying, the other process keeps its state, and `turn` is unchanged.

b) `Enter(i)`: process `i` is trying, and either the other process is non-critical, or the other process is trying too and it is the turn of `i`. Moreover, in the next state, `st` remains the same except that `i` is critical, and the next turn is that of the other process.

c) `Leave(i)`: process `i` is critical and, in the next state, `st` remains the same except that `i` is non-critical, while `turn` is unchanged.

d) `Next`: some process `i` of `Procs` takes a `Try(i)`, an `Enter(i)` or a `Leave(i)` step.

e) `Safety`: processes `1` and `2` are not both critical.

Hint 1. The command `TLA+: Parse module` checks that the module parses correctly.

Hint 2. For d), use the existential quantifier of Exercise 14.

```
---- MODULE MutualExclusion ----
EXTENDS Naturals
VARIABLES st, turn

Procs     == {1, 2}
Other(i)  == IF i = 1 THEN 2 ELSE 1
vars      == << st, turn >>

Init == /\ st = [i \in Procs |-> "n"]
        /\ turn = 1

(* a) *)
Try(i) == /\ st[i] = "n"
          /\ st' = [st EXCEPT ![i] = "t"]
          /\ UNCHANGED turn

(* b) *)
Enter(i) == _____________________________________________________
            _____________________________________________________
            _____________________________________________________
            _____________________________________________________
            _____________________________________________________
```

```
                ____________________________________________________

(* c) *)
Leave(i) == ____________________________________________________
                ____________________________________________________
                ____________________________________________________

(* d) *)
Next == ____________________________________________________

Spec == Init /\ [][Next]_vars

(* e) *)
Safety == ____________________________________________________
====
```

Once the module is complete, save the configuration below as `MutualExclusion.cfg`, select `MutualExclusion.tla`, and run `TLA+: Check model with TLC` to check that `Safety`, as you wrote it, holds. If TLC reports a violation, check your approach again.

```
SPECIFICATION Spec
INVARIANT Safety
```

Exercise 18 (Properties in TLA$^+$). Three more constructs are needed to write the remaining properties of the MEP:

- The formula `P ~> Q` ("`P` leads to `Q`") abbreviates `[](P => <>Q)`: whenever `P` holds, `Q` eventually holds [4].
- The formula `ENABLED A` is true in a state exactly when the action `A` can be taken from the state [4].
- The formula `WF_vars(A)`, *weak fairness* of `A`, says that if `A` becomes enabled for ever, an `A` step eventually occurs [4].

Match each remaining property of Section 7, and the fairness assumption, to its formula by writing its letter in the brackets.

Property	**Formula**
a) Liveness	( ) `\A i \in Procs : (st[i] = "n" => ENABLED Try(i))`
b) Non-blocking	( ) `\A i \in Procs : WF_vars(Enter(i)) /\ WF_vars(Leave(i))`
c) No strict sequencing	( ) none of these formulas
d) Fairness of `Enter` and `Leave`	( ) `\A i \in Procs : (st[i] = "t" ~> st[i] = "c")`

Exercise 19 (Non-blocking).

Part 1 — *LTL or CTL?*. For process 1, non-blocking says: for every state in which process 1 is non-critical, there is a successor state in which it is trying. Write n_1 for `st[1] = "n"` and t_1 for `st[1] = "t"`. Can this property be written in LTL? Write it in CTL.

Part 2 — *Checking it.* In the module, define `NonBlocking` using the formula you matched to non-blocking in Exercise 18. Add `INVARIANT NonBlocking` to the configuration. Save both files and choose *Check again*. Does `NonBlocking` hold?

Part 3 — *More steps ahead.* The formula `ENABLED` expresses $\exists\bigcirc$. The CTL property $\forall\Box(\exists\Diamond\,\mathsf{restart})$, "from every state it is possible to reach a restart state", cannot be written in TLA$^+$. Mark the right explanation with a tick (✓).

( ) TLA$^+$ has no operator for "eventually".

( ) A TLA$^+$ property must hold on every run; it cannot say that *some* run from a state reaches restart, and `ENABLED` looks only one step ahead.

( ) No CTL property can be written in TLA$^+$.

( ) The property would need infinitely many states.

Exercise 20 (Liveness needs fairness). In `MutualExclusion.tla`, define `Liveness` using the formula you matched to liveness in Exercise 18.

Part 1 — *Invariant or property.* Add "`INVARIANT Liveness`" to the configuration, save both files and choose *Check again*. Read the error that TLC reports. Why can `Liveness` not be an invariant? Mark the right explanation with a tick (✓).

( ) The formula `Liveness` contains primed variables.

( ) An invariant is checked one state at a time, but whether `Liveness` holds depends on the states that come later in the run.

( ) A configuration may contain only one invariant.

( ) The formula `Liveness` is false, so TLC refuses to check it.

Part 2 — *Without fairness.* A formula about whole runs is checked with `PROPERTY`. Replace "`INVARIANT Liveness`" with

```
PROPERTY Liveness
```

Save both files and choose *Check again*. TLC reports that `Liveness` is violated. Open the counterexample and read its last two states. Why does `Liveness` fail? Mark the right explanation with a tick (✓).

( ) Both processes become critical at the same time.

( ) When both processes are trying, neither can enter.

( ) When both processes are trying, the specification `Spec` allows runs to take only stuttering steps, so neither process ever enters and the run violates `Liveness`.

( ) TLC stops exploring after four states.

Part 3 — *With fairness.* Define `Fairness` using the formula you matched to the fairness assumption in Exercise 18. Then replace `Spec` by:

```
Spec == Init /\ [][Next]_vars /\ Fairness
```

Save the module and choose *Check again*. The property `Liveness` should hold now. Mark the right explanation with a tick (✓).

a) Why does `Liveness` hold now?

( ) The formula `Fairness` makes the processes enter in strict turns.

( ) The formula `Fairness` removes all stuttering steps.

( ) The formula `Fairness` rules out the runs in which `Enter(i)` or `Leave(i)` stays enabled for ever but never happens, such as the run of the previous part.

b) Why didn't we add a fairness condition on `Try`?

( ) The formula `WF_vars(Try(i))` would force a non-critical process to request the critical section eventually, even if it does not want it. That is not part of the original description of the problem.

( ) The action `Try(i)` is never enabled.

( ) The formula `Liveness` does not mention `Try`, so TLC would reject the fairness condition.

Exercise 21 (No strict sequencing, via a history variable). No strict sequencing says that there is *some* run in which process 1 is critical twice, with no critical state of process 2 in between. An LTL formula cannot say "there is a run", so Huth and Ryan check the complement, *strict sequencing*: "all paths having a c_1 period which ends cannot have a further c_1 state until a c_2 state occurs", written with the weak until W as

$$\mathrm{G}\big(c_1 \to c_1 \,\mathrm{W}\, (\neg c_1 \wedge (\neg c_1 \,\mathrm{W}\, c_2))\big).$$

If strict sequencing is false, that is, if some run violates it, then no strict sequencing is true [3]. TLA$^+$ has no until operator [4]. Instead, we add *history variables*: variables that record what has happened, and never influence the protocol.

Part 1 — *Extending the model.* We add two history variables: `last`, the process that entered the critical section last (`0` before any process has entered), and `twice`, saying whether the process that entered last also entered the time before. The module shown below is `MutualExclusion` extended with them. Complete the missing parts.

```
---- MODULE MutualExclusion ----
EXTENDS Naturals
VARIABLES st, turn, last, twice

Procs     == {1, 2}
Other(i)  == IF i = 1 THEN 2 ELSE 1
(* Hint: The tuple of all 4 variables. *)
vars      == ____________________________________

Init == /\ st = [i \in Procs |-> "n"]
        /\ turn = 1
        /\ last = 0
        /\ twice = FALSE

Try(i) == /\ st[i] = "n"
          /\ st' = [st EXCEPT ![i] = "t"]
          (* Hint: what Try(i) leaves unchanged. *)
```

```
          /\ UNCHANGED ____________________________

Enter(i) == /\ st[i] = "t"
            /\ \/ st[Other(i)] = "n"
               \/ /\ st[Other(i)] = "t"
                  /\ turn = i
            /\ st' = [st EXCEPT ![i] = "c"]
            /\ turn' = Other(i)
            (* Update the value of last' *)
            /\ ____________________________
            (* Hint: twice' is a Boolean expression. *)
            /\ ____________________________

Leave(i) == /\ st[i] = "c"
            /\ st' = [st EXCEPT ![i] = "n"]
            (* Hint: what Leave(i) leaves unchanged. *)
            /\ UNCHANGED ____________________________

Next == \E i \in Procs : Try(i) \/ Enter(i) \/ Leave(i)

Fairness == \A i \in Procs : WF_vars(Enter(i)) /\ WF_vars(Leave(i))
Spec     == Init /\ [][Next]_vars /\ Fairness

Safety      == ~(st[1] = "c" /\ st[2] = "c")
NonBlocking == \A i \in Procs : (st[i] = "n" => ENABLED Try(i))
Liveness    == \A i \in Procs : (st[i] = "t" ~> st[i] = "c")
====
```

Part 2 — *Checking it.* Strict sequencing says that no process ever enters twice in a row. Add the definition below to the module and `INVARIANT StrictSequencing` to the configuration.

```
StrictSequencing == ~twice
```

Save both files, and choose *Check again*. TLC reports a violation. Read the counterexample one transition at a time, and mark the right answer to each question with a tick (✓).

a) What happens in the counterexample?

- ( ) Processes 1 and 2 enter the critical section alternately.
- ( ) Process 1 enters, leaves and enters again, while process 2 stays non-critical.
- ( ) Both processes are critical at the same time.

b) What does the violation tell us?

- ( ) The protocol is wrong and must be fixed.
- ( ) TLC cannot check this kind of property.
- ( ) Strict sequencing is false. Thus, "no strict sequencing holds": the processes need not enter in strict sequence.

8 Case study: a queue inside a lock manager

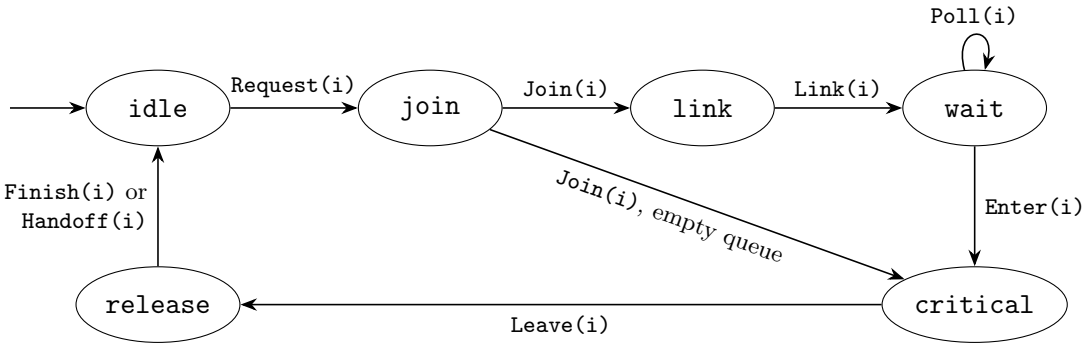
Since 2011, engineers at Amazon Web Services have written TLA$^+$ specifications for systems they use [6, 7]. For instance, two rows from a table in their report are shown below [7]:

System	Components	Benefit
DynamoDB	Replication and group-membership system	Found three bugs requiring traces of up to 35 steps
Internal distributed lock manager	Lock-free data structure	Improved confidence though failed to find a liveness bug, as liveness not checked

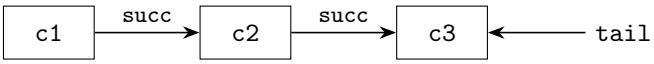
Their reports do not give the algorithms behind those bugs. Thus, to simulate Amazon's lock manager problem, our lecture example will use a queue lock adapted from Mellor-Crummey and Scott's *MCS lock* [8], with a modified release rule. The idea is that several clients use a lock to take turns into a critical section. Each client has its own queue node. A client records its predecessor when it joins the queue, then writes a successor link into that predecessor's node. The predecessor can pass the lock, by setting its entry in `ready`. For modelling, let us fix that each TLA+ action in this exercise represents an atomic step. Another client may act between any two steps. Clients may request the lock repeatedly. Let us assume too that there are no crashes and no reordering of these atomic memory operations.

The model records the following:

- A shared variable, `tail`, that names the last client to join (`NoOne`, with an empty queue).
- The entry `pred[i]` notes the predecessor that client `i` found when joining.
- The entry `succ[i]` names the successor that has linked itself to client `i` (could be `NoOne`).
- The entry `ready[i]` records if client `i` has been signalled to enter the critical section.
- The entry `phase[i]` records client `i`'s phase: `idle`, `join`, `link`, `wait`, `critical`, or `release`.



(a) The phases `phase[i]` of one client `i`, and the actions that move it between them.



(b) A queue of three clients: `c1` holds the lock, `c2` and `c3` wait behind it, and `tail` names the last one.

Figure 3: The queue lock.

Exercise 22 (Building the queue). Save the skeleton below as `LockManager.tla`. The constant `Clients` will get its value from the configuration (see below). Complete the missing actions from the descriptions below. The action `Request` is already provided. Keep each action separate, as specified above.

Note: The symbol `#` is TLA$^+$'s $\neq$ [4]. The given formula `Fairness` is weak fairness of each client's working steps, as in Exercise 20. It does not force an idle client to request the lock [4].

The given action `Poll(i)` lets a waiting client read `ready[i]` again; without it, TLC would report a state in which every client waits as a deadlock.

a) The action `Request(i)` requires client i's current phase to `idle`; resets the (next) functions `succ` and `ready`, for that client only, to `NoOne` and `FALSE`, respectively; it updates the next phase by mapping the client to `join`, and leaves `tail` and `pred` unchanged.

b) The action `Join(i)` requires client i's current phase to be `join`; updates the (next) function `pred`, for that client only, to the current `tail`; sets the next `tail` to i; updates the next phase by mapping the client to `critical` if the current `tail` is `NoOne`, and to `link` otherwise; and leaves `succ` and `ready` unchanged.

c) The action `Link(i)` requires client i's current phase to be `link`; updates the (next) function `succ`, for the client's predecessor `pred[i]` only, to i; updates the next phase by mapping the client to `wait`; and leaves `tail`, `pred` and `ready` unchanged.

d) The action `Enter(i)` requires client i's current phase to be `wait` and `ready[i]` to be `TRUE`; updates the next phase by mapping the client to `critical`; and leaves `tail`, `pred`, `succ` and `ready` unchanged.

e) The action `Finish(i)` requires client i's current phase to be `release` and `succ[i]` to be `NoOne`; sets the next `tail` to `NoOne` if the current `tail` is i, and keeps the current `tail` otherwise; updates the next phase by mapping the client to `idle`; and leaves `pred`, `succ` and `ready` unchanged.

f) The action `Handoff(i)` requires client i's current phase to be `release` and `succ[i]` not to be `NoOne`; updates the (next) function `ready`, for the successor `succ[i]` only, to `TRUE`; updates the next phase by mapping the client to `idle`; and leaves `tail`, `pred` and `succ` unchanged.

```
---- MODULE LockManager ----
EXTENDS FiniteSets, Naturals
CONSTANT Clients
VARIABLES tail, pred, succ, ready, phase

NoOne == "none"
vars == <<tail, pred, succ, ready, phase>>

Init == /\ tail = NoOne
        /\ pred = [i \in Clients |-> NoOne]
        /\ succ = [i \in Clients |-> NoOne]
        /\ ready = [i \in Clients |-> FALSE]
        /\ phase = [i \in Clients |-> "idle"]

(* a) *)
Request(i) == /\ phase[i] = "idle"
              /\ succ' = [succ EXCEPT ![i] = NoOne]
              /\ ready' = [ready EXCEPT ![i] = FALSE]
              /\ phase' = [phase EXCEPT ![i] = "join"]
              /\ UNCHANGED <<tail, pred>>

(* b) *)
Join(i) == _____________________________________________________
           _____________________________________________________
           _____________________________________________________
```

```
            _____________________________________________________
            _____________________________________________________
            _____________________________________________________

(* c) *)
Link(i) == _____________________________________________________
           _____________________________________________________
           _____________________________________________________
           _____________________________________________________

(* d) *)
Enter(i) == ____________________________________________________
            ____________________________________________________
            ____________________________________________________
            ____________________________________________________

Leave(i) == /\ phase[i] = "critical"
            /\ phase' = [phase EXCEPT ![i] = "release"]
            /\ UNCHANGED <<tail, pred, succ, ready>>

(* e) *)
Finish(i) == ___________________________________________________
             ___________________________________________________
             ___________________________________________________
             ___________________________________________________
             ___________________________________________________

(* f) *)
Handoff(i) == __________________________________________________
              __________________________________________________
              __________________________________________________
              __________________________________________________
              __________________________________________________

Poll(i) == /\ phase[i] = "wait"
           /\ ~ready[i]
           /\ UNCHANGED vars

Work(i) == Join(i) \/ Link(i) \/ Enter(i) \/ Leave(i)
           \/ Finish(i) \/ Handoff(i)
Next == \E i \in Clients : Request(i) \/ Work(i) \/ Poll(i)
Fairness == \A i \in Clients : WF_vars(Work(i))
Spec == Init /\ [][Next]_vars /\ Fairness
====
```

The expression `{i \in S : P}` is the set of the elements `i` of `S` for which `P` holds [4], so the set `Inside` contains the clients whose phase is `critical`. The operator `Cardinality`, from `FiniteSets`, counts the elements of a finite set [4]. Before the module's closing line of equals signs, add the definitions below and complete the invariant: *at most one client is in the critical section.*

```
Inside == {i \in Clients : phase[i] = "critical"}
MutualExclusion == _____________________________________________
```

Save the configuration below as `LockManager.cfg`. The identifiers `c1`, `c2` and `c3` are three distinct model values selected for this run [4]. Select `LockManager.tla` and run `TLA+: Check model with TLC` to check the property you just wrote.

```
SPECIFICATION Spec
CONSTANT Clients = {c1, c2, c3}
INVARIANT MutualExclusion
```

Exercise 23 (Does every requester enter?).

Part 1 — *A second property.* Write the formula `Progress` below. It quantifies over every client `i` of `Clients`; for that client, it states that a state in which client *i*'s phase is `join` *leads to* (Exercise 18) a state in which client *i*'s phase is `critical`. Insert the definition before the closing line of `LockManager.tla`, then add `PROPERTY Progress` to the configuration, keeping the invariant. Save both files and choose *Check again.*

```
Progress == _____________________________________________
            _____________________________________________
```

Part 2 — *Following the counterexample.* TLC reports that `Progress` is violated: its counterexample is a run in which a client requests the lock and never becomes `critical`. Each state of the run is shown with the action that produced it, and the run ends in a state that repeats for ever (*Stuttering*). Read the run backwards, from that final state, to find out why no client enters. Answer each item from your counterexample. Your counterexample may differ from the one in the solution: the client names and the order of independent steps can change between runs of TLC.

Hint 1. For b) to d): one action of `q` writes `tail`, and a different, later action of `q` writes `succ[r]`. Find both in the definitions of `Join` and `Link`.

Hint 2. Suppose `Finish(r)` happens between those two actions of `q`. What does `r` see in `succ[r]`, and in `tail`?

a) In the final state, complete:

The phase of every client is _____.
The value of `ready` for every client is _____.

b) Find the last `Finish` step of the run, and let's call the client that takes it `r`. Just before that step, complete:

The successor entry `succ[r]` is _____.
Is the comparison `tail = r`, `TRUE` or `FALSE`? _____.

c) Find the client `q` whose predecessor entry `pred[q]` is `r`. Just before the `Finish` step of `r`, complete:

The phase `phase[q]` is _____.

d) Order these three actions as they occur in the run: `Link(q)`, `Finish(r)`, `Join(q)`.
_____.

e) Mark the explanation consistent with the final state and with your answers above:

( ) A client stays in `critical` for ever, although its `Leave` action stays enabled.

( ) Every client waits with `ready` false. Only a `Handoff` step sets an entry of `ready` to true, and it needs a client in `release`. No client is in `release`, so no `Handoff` step can happen.

( ) A waiting client has `ready` true, but the scheduler ignores its enabled `Enter` action for ever.

Exercise 24 (A repair or a different model?).

Part 1 — *Combining two actions.* A colleague proposes to combine `Join` and `Link` into a single action, `JoinTogether(i)`, so that no other client can act between them. Write it from its description:

- The action `JoinTogether(i)` requires client i's current phase to be `join`; updates the (next) function `pred`, for that client only, to the current `tail`; sets the next `tail` to i; updates the (next) function `succ`, for the current `tail` only, to i if the current `tail` is not `NoOne`, and leaves `succ` unchanged otherwise; updates the next phase by mapping the client to `critical` if the current `tail` is `NoOne`, and to `wait` otherwise; and leaves `ready` unchanged.

```
JoinTogether(i) == _____
                   _____
                   _____
                   _____
                   _____
                   _____
                   _____
                   _____
```

Insert this definition just before `Work(i)`. Temporarily replace `Join(i) \/ Link(i)` in `Work(i)` with `JoinTogether(i)`. Keep the original `Finish`, the fairness condition, the invariant, and `PROPERTY Progress`. Save and check again. Mark the explanation of the result:

( ) The check now establishes that the original implementation, with two separate steps, satisfies `Progress`.

( ) The check covers a model in which another client cannot act between updating the tail and writing the link.

( ) The check succeeds because the fairness condition became stronger.

Part 2 — *Keeping the original steps.* Restore `Join(i) \/ Link(i)` in `Work(i)` and remove the temporary `JoinTogether` definition. Change only `Finish`: it may return to `idle` with no successor link only when the shared tail *still names the releasing client*. The comparison with `tail` and the update of `tail` stay in the same step. Write the new `Finish` from its description:

- The action `Finish(i)` requires client i's current phase to be `release`, `succ[i]` to be `NoOne`, and the current `tail` to be i; sets the next `tail` to `NoOne`; updates the next phase by mapping the client to `idle`; and leaves `pred`, `succ` and `ready` unchanged.

```
Finish(i) == _____________________________________________
             _____________________________________________
             _____________________________________________
             _____________________________________________
             _____________________________________________
             _____________________________________________
```

Save and check both properties again. Then revisit the intermediate state from Exercise 23: the tail has changed, but the successor link has not yet been written. Mark what the revised model permits there:

( ) The releasing client returns to `idle` immediately, and fairness eventually makes the waiting client's `ready` true.

( ) The releasing client stays in `release`. After the joining client takes `Link`, the releasing client can take `Handoff`, which sets the joining client's entry in `ready` to true.

( ) The joining client discards its predecessor and enters the critical section directly.

Part 3 — *One more client (optional).* Change the configured set to `{c1, c2, c3, c4}` and recheck the repaired model, including `PROPERTY Progress`. This run may take longer. Mark the conclusion supported by a successful check:

( ) Both properties hold in the checked four-client model, under its fairness and atomicity assumptions.

( ) Both properties have now been proved for every finite number of clients.

( ) Amazon's implementation has now been checked.

9 Summary: Spin and TLA$^+$ side by side

Exercise 25 (Which tool?). For each description, mark with a tick (✓) the tool, or tools, it applies to. Some descriptions apply to both. An example is provided.

Description	Spin	TLA$^+$
Models are written in a language of processes.	(✓)	( )
Models are written as mathematics: sets, functions and formulas.	( )	( )
One arrow of the LTS is one option of a `do` loop.	( )	( )
One arrow of the LTS is one disjunct of `Next`.	( )	( )
An atomic proposition is a `#define` macro.	( )	( )
An atomic proposition is a formula written directly in the property.	( )	( )
Temporal formulas can use "next" and "until".	( )	( )
A state in which no step is possible is reported as an error.	( )	( )
A run may stutter for ever, unless fairness rules it out.	( )	( )
Fairness is written as part of the specification.	( )	( )
A property can say that a step is possible (`ENABLED`).	( )	( )
The checker is a C program generated from the model.	( )	( )
What to check is listed in a separate configuration file.	( )	( )
When a property fails, it shows a counterexample.	( )	( )

Both tools are presented by their authors for concurrent and distributed systems [1, 5]. Amazon chose TLA$^+$ against their own requirements: handling "very large, complex or subtle problems", minimising "cognitive burden", and a "high return on investment" [6].

References

[1] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[2] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[3] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Second edition. Cambridge University Press, 2004.

[4] Leslie Lamport. *Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[5] Leslie Lamport. *The TLA$^+$ Home Page*. https://lamport.azurewebsites.net/tla/tla.html.

[6] Chris Newcombe. Why Amazon Chose TLA$^+$. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ 2014)*, LNCS 8477, pages 25–39. Springer, 2014.

[7] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker and Michael Deardeuff. How Amazon Web Services Uses Formal Methods. *Communications of the ACM*, 58(4):66–73, April 2015.

[8] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991. https://www.cs.rice.edu/~johnmc/scalable_synch/tocs91.pdf.