# SWEN90004
# Modelling Complex Software Systems

## Processes in FSP

Nic Geard

Lecture Con.05

Semester 1, 2019

# Introduction

So far, we've seen how to create multi-process/thread programs in Java, and explored some of the problems that we encounter when writing concurrent programs. These problems do not occur on every execution of the program, even if the input remains the same, owing to the inherent non-determinism in concurrent programs.

When dealing with large code bases, identifying, locating, and removing concurrency problems can be a nightmare. The size of the system and the number of possible interleavings and synchronisations become so large that it is difficult to even understand a single problem, isolate it, figure out how to fix it, or prevent it in the first place. So far we studied low-level models for dealing with mutual exclusion and monitoring, but these are often too low-level to be of use for analysing whether a system can deadlock.

# Formal modelling: Process algebra

In this section of subject, we will move from low-level models to more abstract, formal models. In particular, we will look at a language called Finite State Processes (FSP), based on the well-known Communicating Sequential Processes (CSP) and Calculus of Communicating Systems (CCS).

The rules for manipulating and reasoning about expressions in these languages is referred to as process algebra (or, sometimes, a process calculus).

# Formal modelling

Our focus will be on FSP. The language has a well-defined syntax and semantics, much the same as programming languages do. In fact, FSP has a semantics that is more rigorously defined than most programming languages.

Consider these two advantages of formal modelling:

- It forces a preciseness in thinking.
- It provides us with the rigour needed to analyse our models, compare them with the physical circumstances of the problem and make trade-offs in a precise way.

Reasoning about interaction at an informal level is like trying to test a program from its comments. No matter how well commented a program is, the comments are written in natural language, and problems of ambiguity and incompleteness arise.

# LTS—an introduction

Our reasons for using FSP instead of, for example, CSP or CCS, are:

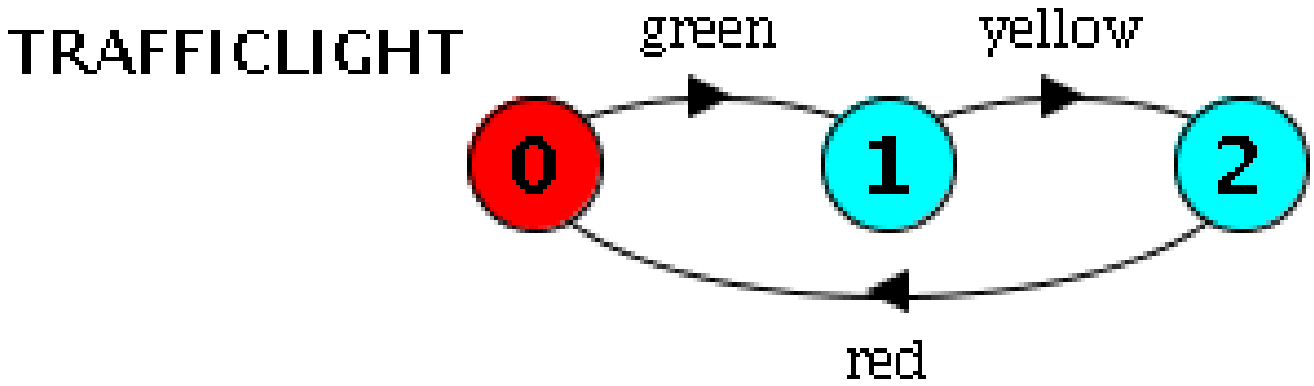
- it has a machine readable syntax;
- the models are finite (the 'F' in FSP);

These mean that we can execute them and exhaustively prove properties about them.

We shall use finite state machines as models of programs. Our particular type of state machine is the labelled transition system (LTS).

# LTS—an introduction

Here is a LTS for a traffic light:




The LTS captures that the light must go `green -> yellow -> red -> green -> yellow -> ...`, etc. The sequence `green-> red` is illegal. In this system, the labels `green`, `yellow`, and `red` represent atomic actions that make indivisible state changes.

# LTS—an introduction

Note that the traffic light system is infinite. The sequence `green -> yellow -> red` continues infinitely, and the process never terminates.

Non-termination is common in concurrent, real-time systems. For example, operating systems are designed to be non-terminating. Even though they can be terminated in practice, and problems such as power outages take them off-line, they must execute indefinitely, so are designed to execute infinitely.

# LTS and real-time systems

The traffic light LTS does not specify any timing for the light. The light may be yellow for exactly two seconds, and red for exactly twenty. However, the LTS only specifies the sequence of legal actions in the traffic light system.

There are formalisms that deal with timing. Variants of LTSs have been invented to deal with timing, but we won't cover these. That is, we shall use FSP only for studying properties related to processes and their synchronisation, not their timing properties.

Our approach is justified by appealing to the purpose of modelling. To model, we must abstract away some detail that we consider less relevant.

# FSP—algebraic representation

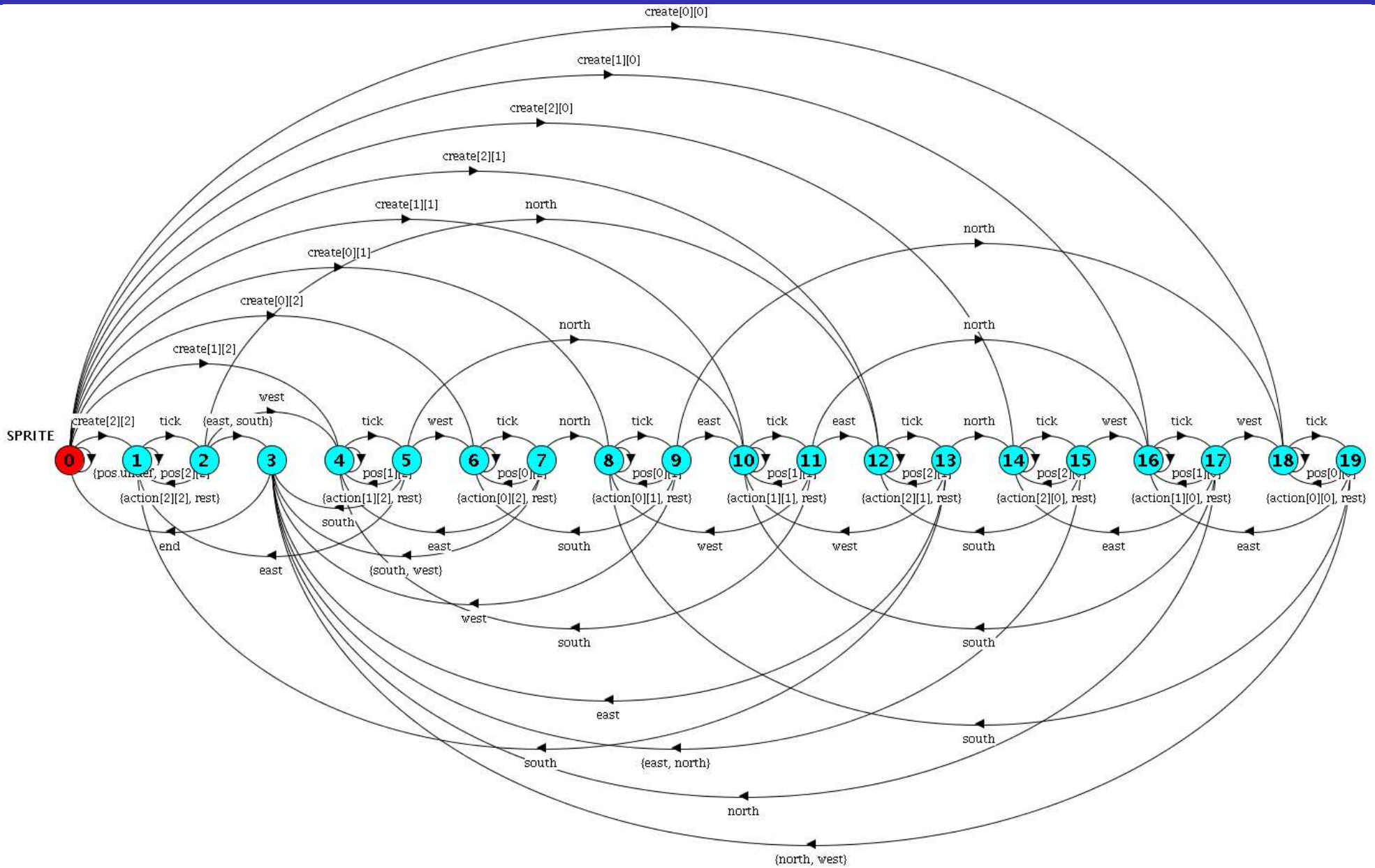
The graphical representation of a system is fine for small, non-trivial systems, but quickly becomes unmanageable and unreadable for any real problems, owing to the number of states and transitions.

For example, the the next slide shows an LTS for the legal moves of a player in a game consisting over a board only 2x2 large.

Consequently, process models are described in an algebraic language called finite state processes (FSP).

Each FSP process model has a corresponding LTS model. The FSP for the player in the game is only 11 statements long.

# Example—the SPRITE process



# FSP—basic concepts

FSP allows the description of one or more processes that operate independently, but which may synchronise at various points. Each process model in FSP consists of a set of atomic actions (the "alphabet") that can occur in that process, and a definition which specifies the legal sequences of atomic actions in the system.

Each process, and how concurrent processes synchronise, is described using a collection of algebraic operators.

We shall use a tool called the LTS analyser (LTSA) to automatically convert FSP models into LTS models, and to automatically analyse properties of LTS models. It is available at `http://www.doc.ic.ac.uk/ltsa/` and on the lab computers.

# FSP—the action prefix

The action prefix operator, "->", is fundamental:

> *"If* `x` *is an action and* `P` *a process then the action prefix* `x -> P` *describes a process that initially engages in action* `x` *and then behaves exactly as described by* `P`*."*

```
TRAFFIC_LIGHT = (green->yellow->red->TRAFFIC_LIGHT).
```

The "->" operator always has an atomic action as the left operand, and a process as the right operand. In this case, `green` is the left operand, and `yellow -> red -> TRAFFIC_LIGHT` is the right operand (the process). `TRAFFIC_LIGHT` is the name of the process. Repetitive behaviour is captured by recursion.

Atomic actions use lower case; process names use upper case.

# FSP—process model definitions

The traffic light behaviour could be modelled using more than one process name; for example:

```
TRAFFIC_LIGHT = GREEN,
GREEN = (green -> YELLOW),
YELLOW = (yellow -> RED),
RED = (red -> GREEN).
```

This definition generates the same LTSA as the one-line definition on the previous slide.

Note the use of "," and "." at the end of lines. The "," indicates that the processes named `GREEN`, `YELLOW`, and `RED` are part of and local to the definition of the process named `TRAFFIC_LIGHT`.

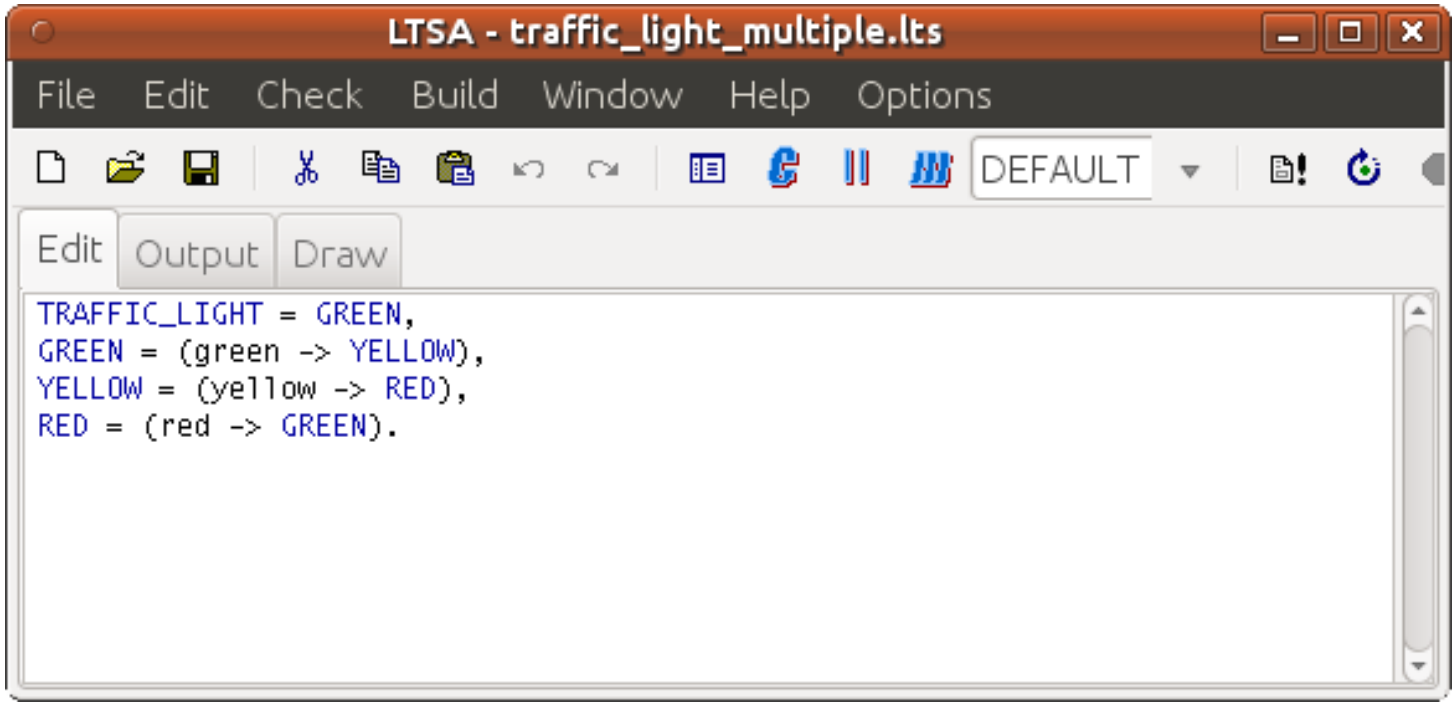
# Using LTSA—editing

Before we go further with FSP operators, we'll take a peek at the LTS analyser (LTSA). We'll look more at this in the workshops, but it is worthwhile playing around with this to get an understanding of FSP and LTS.

The tool from `http://www.doc.ic.ac.uk/ltsa/`, and follow the instructions for running it. The LTSA website contains a reference manual as well as some examples.

# Using LTSA—editing

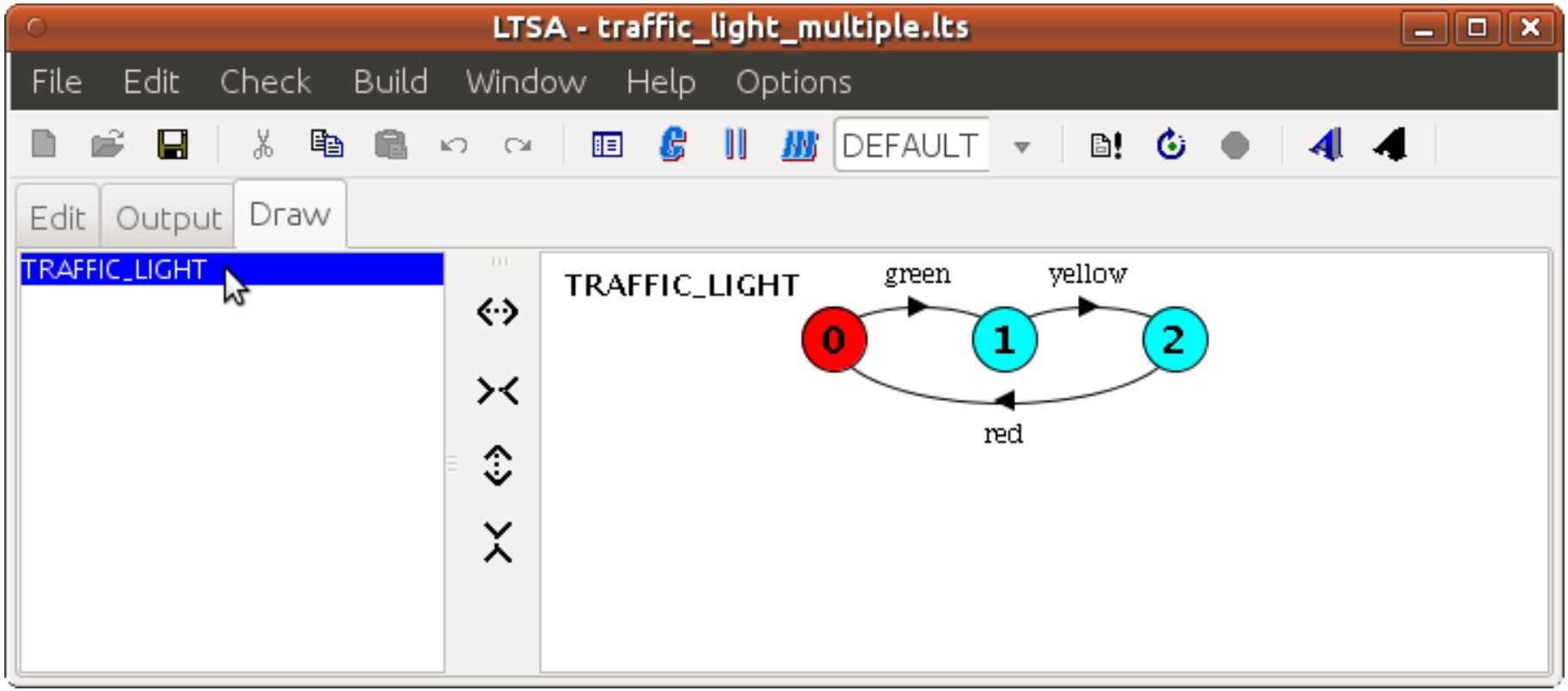
The first screen will resemble this:




We have typed the definition of `TRAFFIC_LIGHT` under the Edit tab. Now do Build → Compile. This will do a syntax check. Try changing some of the syntax to generate a syntax error.

# Using LTSA—drawing

Click on the Draw tab, and then on `TRAFFIC_LIGHT` in the left pane. LTSA will display the LTS for the FSP definition.



# Using LTSA—drawing

Modify the FSP definition of `TRAFFIC_LIGHT` by inserting a `yellow` action in between the `red` and `green` actions, which is consistent with traffic lights in some countries.

Compile this and look at the new LTS.

# Using LTSA—animating

To interact with your process model, go to the menu Check → Run → DEFAULT. You will see a window like the one on the left:



The `green` action is the only action that is enabled, because in the definition of the traffic light process, it is the first action. You can "run" the green action by clicking on the "green" label.

# Using LTSA—animating



When an action is run, it appears in the left pane of the window, as shown in the right screen shot. In this case, we have run `green -> yellow -> red -> green`, and now `yellow` is the only enabled action. (Under the `Draw` tab of the main window, the latest transition and the current state are both highlighted.)

# Using LTSA—animating

The animation mode allows us to "test" our model by interacting with it and exploring the traces that it generates.

Modify the traffic light process model and animate it to see the differences.

# FSP—choice

The choice operation, "|" is used to describe a process that can execute more than one possible sequence of actions.

> *"If* `x` *and* `y` *are actions, then* `(x -> P | y -> Q)` *describes a process which initially engages in either of the actions* `x` *or* `y`. *After the first action has occurred, the subsequent behaviour is described by* `P` *if the first action was* `x` *and* `Q` *if the first action was* `y`*."*

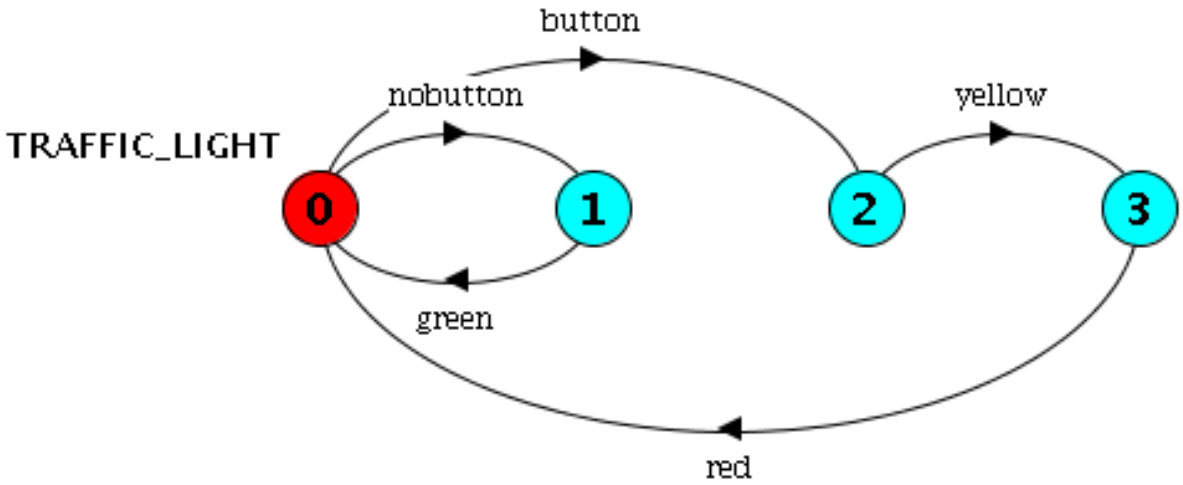
# FSP—choice

This describes a traffic light with a button for a pedestrian, which turns the light red. If the button is not pushed, the light remains green:

```
TRAFFIC_LIGHT = (button -> YELLOW | none -> GREEN),
GREEN =  (green -> TRAFFIC_LIGHT),
YELLOW = (yellow -> RED),
RED = (red -> TRAFFIC_LIGHT).
```

It may seem somewhat strange to model an action that does not really occur (`none`), but this is simply used to describe that `button` has not occurred.

# FSP—choice




The initial state has two outgoing transitions: one labelled `button`, one `none`. At the start of the process, two actions are enabled. If the pedestrian button is pressed, the light goes yellow, then red, and finally back to waiting for a button to be pressed. When a button is not pressed, the light remains green.

From the initial state, only one transition can be chosen. Who makes the choice?

# FSP—input and output

In this example, the choice is made by the environment (a pedestrian), not the process itself.

This example may also raise a question: What is the input to the process, and what is the output?

FSP makes no distinction make between input and output actions. `button` is an input, while `green` is an output, but FSP does not distinguish these semantically—they are both just actions.

However, usually actions that form part of a choice are considered inputs (in this case, `button` and `none`), and actions that offer no choice are outputs (`green`, `yellow`, and `red`).

# FSP—non-deterministic choice

To complicate matters regarding input, output, and choice, FSP allows non-deterministic choice.

> *"Process* `(x -> P | x -> Q)` *describes a process that engages in* `x` *and then behaves as* `P` *or* `Q`*."*

Note that `x` is the prefix in both options of the choice. In this instance, the choice is made by the process, not the environment. Therefore, `x` could be an input from the environment, but the choice of `P` or `Q` is not controlled by the environment.

# FSP—non-deterministic choice

In the following example, the traffic light process has a (deterministic) choice between `button` and `none`, but if `button` occurs then the process can either go immediately yellow, or wait for one green "tick", and then go yellow. The choice is up to the process itself.

```
TRAFFIC_LIGHT = ( button -> YELLOW
                | button -> green -> YELLOW
                | none -> GREEN
                ),
GREEN =  (green -> TRAFFIC_LIGHT),
YELLOW = (yellow -> RED),
RED = (red -> TRAFFIC_LIGHT).
```

# FSP—indexed processes

To model a process that can take multiple values, indexed processes can be used. In an indexed process, variables can be used to increase the expressive power of FSP.

Consider a buffer that can contain a single value. The value is input into the buffer, and can be then output. Values range from 0 to 3:

```
BUFF = (in[i:0..3] -> out[i] -> BUFF).
```

This is equivalent to:

```
BUFF = (in[0] -> out[0] -> BUFF
       |in[1] -> out[1] -> BUFF
       |in[2] -> out[2] -> BUFF
       |in[3] -> out[3] -> BUFF
       ).
```

Try these in LTSA—look at their respective LTS representations.

# FSP—constants and ranges

To improve the maintainability and to generalise models, constants can be declared. Constants can take on the integer values only. In addition, ranges, which are finite ranges of integers, can be used:

```
const N = 3
range T = 0..N

BUFF = (in[i:T] -> STORE[i]),
STORE[i:T] = (out[i] -> BUFF).
```

In the above example, the maximum value of 3 is declared as a constant `N`, and the range `T` is used in two places. If the maximum value increases to 5, we need only change `N`.

# FSP—guarded actions

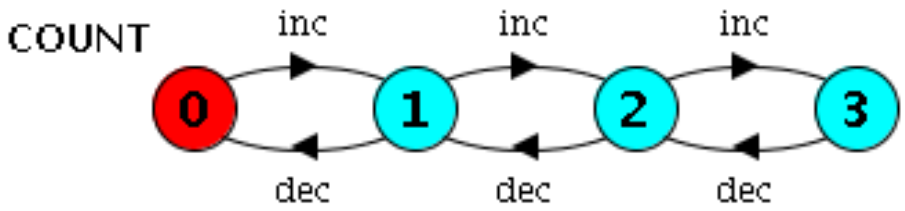
A guarded action allows a context condition to be added to options in a choice.

> *"The choice* `(when B x -> P | y -> Q)` *means that when the guard* `B` *is true, then the actions* `x` *and* `y` *are both eligible to be chosen, otherwise if* `B` *is false, then action* `x` *cannot be chosen."*

```
COUNT (N=3)      = COUNT[0],
COUNT[i:0..N] = (when(i<N) inc->COUNT[i+1]
                |when(i>0) dec->COUNT[i-1]
                ).
```

# FSP—guarded actions

The corresponding LTS is:



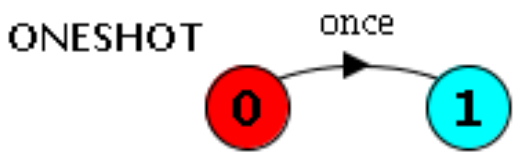
# FSP—the STOP process

The STOP process is a special, pre-defined process that engages in no further actions. It is used for defining processes that terminate.

For example, consider the following FSP model, which executed a single action and then terminates:

```
ONESHOT = (once -> STOP).
```

The corresponding LTS is:



# FSP—the STOP process

The definition of STOP is:

```
const False = 0
P = (when (False) doanything -> P).
```

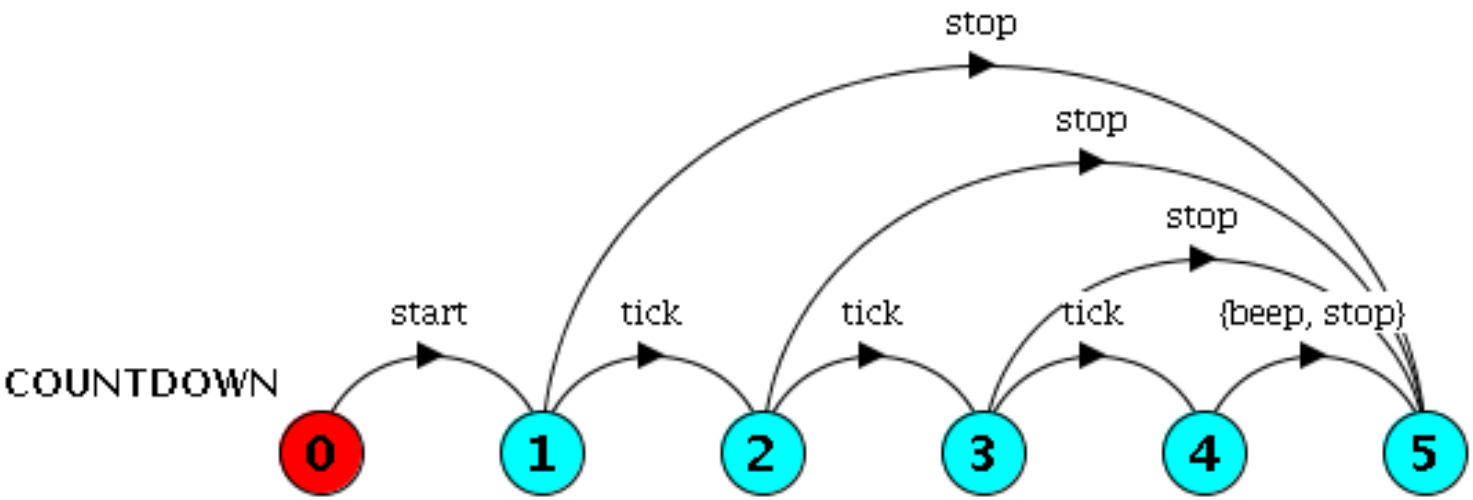
That is, when `False` is true (never) do any action (the action itself does not matter), and then repeat.

**Exercise:** In the LTSA tool, write the definition for `STOP`, compile it, and look at the corresponding LTS model in the Draw tab.

# Example: a countdown timer

The following is an example of a timer that counts down from 3, and then beeps, or can be stopped at any point before the beep.

```
COUNTDOWN (N=3) = (start->COUNTDOWN[N]),
COUNTDOWN[i:0..N] =
    (when(i>0)  tick->COUNTDOWN[i-1]
    |when(i==0) beep->STOP
    |stop->STOP
    ).
```



# References / Recommended reading


- J. Magee and J. Kramer, *Concurrency: State Models and Java Programs*, 2nd edition, John Wiley and Sons, 2006.