# Embedded Systems Specification and Design
# Model-based Design and Verification

David Kendall

## Bank Account: A simple 2-process algorithm

Let's model a simple system in which 2 processes withdraw money from a bank account

The bank account is modelled just by its `balance`, which is initially £100

Each process withdraws £10 and then stops. The final balance should be £80

```
--algorithm BankAccount
    variable balance = 100;

    process Withdraw10 \in (1..2)
      variable current = 0;
    begin
s1:   current := balance;
s2:   current := current - 10;
s3:   balance := current;
    end process
end algorithm
```

## Notes on the Pluscal Bank Account

A multi-process algorithm is introduced and concluded in the usual way

```
--algorithm SomeName

end algorithm
```

Processes are introduced with the keyword `process`. Each process has a name (not necessarily unique) and an identifier (unique), taken from the same set as all other processes. A process can introduce local variables, e.g.

```
process Withdraw10 \in (1..2)
variable

begin

end process
```

This introduces 2 processes called `Withdraw10` and identified by elements in the set $\{1, 2\}$

# TLA$^+$ translation of Pluscal Bank Account

```
\* BEGIN TRANSLATION
VARIABLES balance, pc, current

vars == << balance, pc, current >>

ProcSet == (1..2)

Init == (* Global variables *)
        /\ balance = 100
        (* Process Withdraw10 *)
        /\ current = [self \in 1..2 |-> 0]
        /\ pc = [self \in ProcSet |-> "s1"]

s1(self) == /\ pc[self] = "s1"
            /\ current' = [current EXCEPT ![self] = balance]
            /\ pc' = [pc EXCEPT ![self] = "s2"]
            /\ UNCHANGED balance

s2(self) == /\ pc[self] = "s2"
            /\ current' = [current EXCEPT ![self] = current[self] - 10]
            /\ pc' = [pc EXCEPT ![self] = "s3"]
            /\ UNCHANGED balance

s3(self) == /\ pc[self] = "s3"
            /\ balance' = current[self]
            /\ pc' = [pc EXCEPT ![self] = "Done"]
            /\ UNCHANGED current

Withdraw10(self) == s1(self) \/ s2(self) \/ s3(self)

Next == (\E self \in 1..2: Withdraw10(self))
           \/ (* Disjunct to prevent deadlock on termination *)
              ((\A self \in ProcSet: pc[self] = "Done") /\ UNCHANGED vars)
\* END TRANSLATION
```

## Notes on the TLA$^+$ translation

The global and local variables are all introduced in the same `VARIABLES` section

A definition, `ProcSet`, is introduced to represent the process identifiers

The local variables are modelled by functions from process identifiers to variable values, i.e. each process has its own copy of the local variables, indexed by process identifier

The *labels*, `s1`, `s2`, and `s3`, are used to identify the actions (steps) of the process. Each step is translated into its own TLA$^+$ operation. The placement of labels defines the *granularity* of the steps and is crucial to the behaviour of processes

The TLA$^+$ translation introduces a local variable, `pc` (program control), for each process, to indicate the current step in its execution

For a process that terminates, a label, `Done`, is introduced to model termination

Each process is modelled as the disjunction of its steps

The `Next` relation is modelled as the disjunction of the processes

## Using TLC to check the BankAccount model

For a bank account with an initial balance of £100, we expect the final balance to be £80, after the completion of 2 withdrawals of £10 each

We can use TLC to check that this is the case for the bank account that we have modelled in Pluscal

Termination of the processes is indicated when the value of `pc` for each process is `Done`

We can define a `BalanceOk` property as follows:

```
BalanceOk == (pc[1] = "Done" /\ pc[2] = "Done") => balance = 80
```

We can use TLC to check if this property is invariantly TRUE. If it is not TLC will show a behaviour, starting in the initial state, that reaches a state in which the alleged invariant is FALSE

**Demo** Show the use of TLC to check this invariant.

## Another synchronisation problem

Create a Pluscal model of a 2 process system comprising processes P1 and P2. P1 tests if global variable y is 0 and, if it is, it updates the value of global variable x to 1. P2 tests if global variable x is 0 and, if it is, it updates the value of global variable y to 1. Is it possible for both x an y to be 1 at the end?

## The Pluscal algorithm

```
--algorithm TwoProcesses
  variable
    x = 0,
    y = 0

  process P1 = "x"
  begin
x1: if y = 0 then
x2:   x := 1
    end if
  end process

  process P2 = "y"
  begin
y1: if x = 0 then
y2:   y := 1
    end if
  end process
end algorithm
```

# The TLA$^+$ translation

```
\* BEGIN TRANSLATION
VARIABLES x, y, pc

vars == << x, y, pc >>

ProcSet == {"x"} \cup {"y"}

Init == (* Global variables *)
        /\ x = 0
        /\ y = 0
        /\ pc = [self \in ProcSet |-> CASE self = "x" -> "x1"
                                        [] self = "y" -> "y1"]

x1 == /\ pc["x"] = "x1"
      /\ IF y = 0
            THEN /\ pc' = [pc EXCEPT !["x"] = "x2"]
            ELSE /\ pc' = [pc EXCEPT !["x"] = "Done"]
      /\ UNCHANGED << x, y >>

x2 == /\ pc["x"] = "x2"
      /\ x' = 1
      /\ pc' = [pc EXCEPT !["x"] = "Done"]
      /\ y' = y

P1 == x1 \/ x2
```

## The TLA$^+$ translation (ctd)

```
y1 == /\ pc["y"] = "y1"
      /\ IF x = 0
            THEN /\ pc' = [pc EXCEPT !["y"] = "y2"]
            ELSE /\ pc' = [pc EXCEPT !["y"] = "Done"]
      /\ UNCHANGED << x, y >>

y2 == /\ pc["y"] = "y2"
      /\ y' = 1
      /\ pc' = [pc EXCEPT !["y"] = "Done"]
      /\ x' = x

P2 == y1 \/ y2

Next == P1 \/ P2
           \/ (* Disjunct to prevent deadlock on termination *)
              ((\A self \in ProcSet: pc[self] = "Done") /\ UNCHANGED vars)
\* END TRANSLATION
```

## What property should be checked?

What property can we check to see if we never have a situation where x and y are both equal to 1?

## What property should be checked?

What property can we check to see if we never have a situation where x and y are both equal to 1?

- `~(x = 1 /\ y = 1)`

## What property should be checked?

What property can we check to see if we never have a situation where x and y are both equal to 1?

- `~(x = 1 /\ y = 1)`
- State this property as an invariant and check it using TLC

## What property should be checked?

What property can we check to see if we never have a situation where x and y are both equal to 1?

- `~(x = 1 /\ y = 1)`
- State this property as an invariant and check it using TLC
- Demo - check the property with TLC

## Mutual exclusion of critical sections

The Bank Account and Two Processes models both illustrate problems of interference that can arise in a concurrent system

We assume that steps of each process might be interleaved in an arbitrary way and see that undesirable results can be delivered

On modern processors, it is straightforward to ensure mutual exclusion but, before the introduction of instructions such as *test-and-set* or *compare-and-swap*, this was not so easy

A number of mutual exclusion algorithms have been proposed that rely only on *memory interlock* — reads and writes to variables at the word-size of the processor are assumed to be atomic

Several of these algorithms (some published in peer-reviewed journals) turned out to be wrong!

## Requirements for a mutual exclusion algorithm

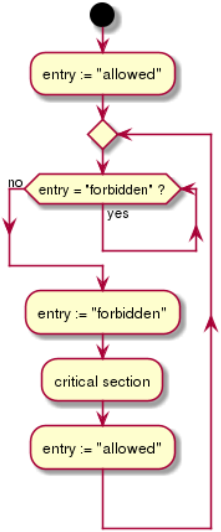
There are 4 key requirements for a satisfactory mutual exclusion algorithm

1. Mutual exclusion must be preserved
2. Deadlock must be avoided
3. Cooperation must not be required between processes beyond following the mutex protocol when entering and leaving a critical section
4. Starvation must be avoided, i.e. any process attempting to enter its critical section must eventually succeed

Much can be learned about the problems of concurrency by investigating mutual exclusion algorithms, even though they are now rarely used in practice

## A failed mutual exclusion algorithm

Two processes execute the same algorithm



## The Pluscal algorithm

```
--algorithm Mutex01
    variable
      entry = "allowed";
    process Proc \in 1..2
    begin
s1:   while TRUE do
s2:     await entry = "allowed";
s3:     entry := "forbidden";
        \* BEGIN CRITICAL SECTION
cs:     skip;
        \* END CRITICAL SECTION
s4:     entry := "allowed"
      end while
    end process
end algorithm
```

## The TLA$^+$ translation

```
\* BEGIN TRANSLATION
VARIABLES entry, pc

vars == << entry, pc >>

ProcSet == (1..2)

Init == (* Global variables *)
        /\ entry = "allowed"
        /\ pc = [self \in ProcSet |-> "s1"]

s1(self) == /\ pc[self] = "s1"
            /\ pc' = [pc EXCEPT ![self] = "s2"]
            /\ entry' = entry

s2(self) == /\ pc[self] = "s2"
            /\ entry = "allowed"
            /\ pc' = [pc EXCEPT ![self] = "s3"]
            /\ entry' = entry
```

## The TLA$^+$ translation (ctd)

```
s3(self) == /\ pc[self] = "s3"
            /\ entry' = "forbidden"
            /\ pc' = [pc EXCEPT ![self] = "cs"]

cs(self) == /\ pc[self] = "cs"
            /\ TRUE
            /\ pc' = [pc EXCEPT ![self] = "s4"]
            /\ entry' = entry

s4(self) == /\ pc[self] = "s4"
            /\ entry' = "allowed"
            /\ pc' = [pc EXCEPT ![self] = "s1"]

Proc(self) == s1(self) \/ s2(self) \/ s3(self) \/ cs(self

Next == (\E self \in 1..2: Proc(self))
```

## What property should be checked?

```
InCS(p) == pc[p] = "cs"
Mutex == ~(InCS(1) /\ InCS(2))
```