SWEN90004
Modelling Complex Software Systems

Wrapping up

Nic Geard

Review Lecture

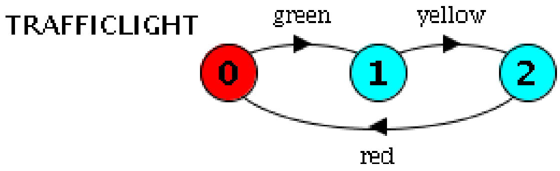
Semester 1, 2019

Topics: FSP and LTS

FSP (Finite State Processes) is an algebraic language for modelling concurrent systems. It comprises processes defined in terms of sequences of atomic actions.

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

LTS (Labelled Transition Systems) is a graphical representation of process execution. Nodes represent system states, while transitions are labelled with the atomic actions that result in system state changing.



Topics: Choice in FSP

The choice operator "|" indicates that one or more action sequences can be executed:

```
P = ( x -> y -> P | a -> b -> P ).
```

Choice can be non-deterministic:

```
Q = (x -> y -> z -> Q | x -> z -> Q ).
```

Topics: Indexed processes and guards

Indexed processes can be used to model a process that can take multiple values:

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

Guards can be used to impose conditions on the execution of action sequences:

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

Topics: Composing processes

```
||P = (Q || R).
```

`||P` is a composite process that models the concurrent execution of processes `Q` and `R`.

Atomic actions from each of the constituent processes will interleave. That is, one action from either process will execute at a time.

Processes that are composed together interact as a result of shared actions. An action which appears in more than one process in a composition must be executed at the same time by all processes that use that shared action.

Topics: Action relabelling

If we wish to indicate that two actions with different names (in two separate processes) should be shared, we can use relabelling when composing the two:

```
P = ( x -> y -> P ).
Q = ( a -> b -> Q ).

||R = ( P || Q/{x/a} ).
```

`P` and `Q` have no shared actions, but renaming `a` to `x` in process `Q` when composing into `||R` means that the first action in each of the constituent processes will be shared, and hence must be executed at the same time.

Topics: Process labelling

Consider a single client:

```
CLIENT = ( call -> wait -> continue -> CLIENT ).
```

We can compose two clients `a` and `b` as follows:

```
||TWO_CLIENTS = ( a:CLIENT || b:CLIENT ).
```

or `N` clients (`c[1], c[2], ..., c[N]`) as follows:

```
||N_CLIENTS(N=3) = ( c[i:1..N]:CLIENT ).
```

Each of the actions of these labelled processes will be prefixed by the process label; eg: `a.call`, `b.call`, `c[1].call`, `c[2].call`, etc.

Topics: Process labelling

If we want to compose multiple clients with a server, we need to also add prefixs to the server actions to ensure they are still shared:

```
||SYSTEM = ( c[i:1..N]:CLIENT || {c[i:1..N]}::SERVER ).
```

Topics: Deadlock

Four necessary and sufficient conditions (the Coffman conditions):

1. serially reusable resources
2. incremental acquisition
3. no preemption
4. wait-for cycle

In an LTS, deadlock corresponds to a node with no outgoing transitions. That is, the FSP model is in a state where it not possible to execute any action.

Topics: Checking safety and liveness in FSP

Two types of property that are of interest in concurrent systems:

- Safety properties: nothing “bad” (eg, deadlock or interference) ever happens during execution
- Liveness properties: something “good” eventually happens during execution (eg, all processes trying to access their critical section eventually do so)

If we model a concurrent process using FSP, we can automatically check that a defined property holds; that is, is true for every possible trace/execution of the model.

Topics: Checking safety in FSP

Safety properties specify desired system behaviour that we want to be maintained by the model.

In FSP, a property is specified in the same way as a process, but using the `property` keyword:

```
property SAFE_ACTUATOR
    = ( command -> respond -> SAFE_ACTUATOR ).
```

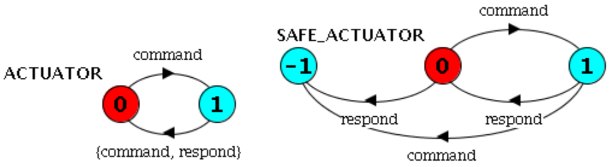
specifies that, whenever a `command` action is executed, a `respond` action should be occur before another `command` action occurs.

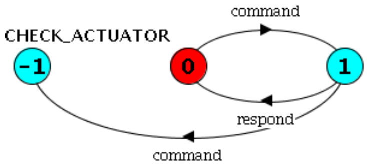
Recall that safety properties must be deterministic.

Topics: Checking safety in FSP

```
ACTUATOR = ( command -> ACT ),
ACT = ( respond -> ACTUATOR | command -> ACTUATOR ).

||CHECK_ACTUATOR = ( ACTUATOR || SAFE_ACTUATOR ).
```





Topics: Checking liveness in FSP

We focused on one type of liveness property – progress – which is used to state that a specified action will eventually execute. (ie, the opposite of starvation).

```
COIN
    = ( toss -> heads -> COIN
      | toss -> tails -> COIN
      ).

progress HEADS = {heads}
progress TAILS = {tails}
```

Topics: Checking liveness in FSP

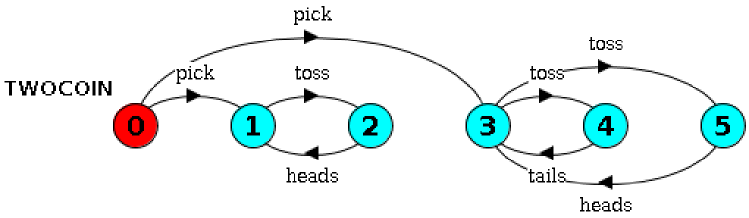
```
TWO_COINS
    = ( pick -> COIN | pick -> TRICK ),
COIN
    = ( toss -> heads -> COIN | toss -> tails -> COIN ),
TRICK
    = ( toss -> heads -> TRICK ).
```

Note that while the `HEADS` progress property will still hold, the `TAILS` progress property will be violated.

The progress property: `progress HEADS_TAILS = {heads, tails}` will hold because only one of the actions in the set needs to occur infinitely often.

Topics: Checking liveness in FSP

Progress properties can be analysed by considering the terminal action sets, identifiable as strongly connected components in the LTS:



Topics: Stress testing

All analyses of FSP models assume a fair choice assumption; ie, that all options in all choices will eventually be taken.

We can also simulate "stressed" systems in which some actions have priority over others:

`P << {a1, ..., aN}` specifies that actions `a1, ..., aN` have a higher priority than all other actions in `P`.

This means that if there is a choice between an action in the set `a1, ..., aN` and an action that is not in this set, the action that is in the set will always be chosen.

`P >> {a1, ..., aN}` specifies that actions `a1, ..., aN` have a lower priority than all other actions in `P`.

Topics: Linear temporal logic (LTL)

Linear temporal logic (LTL) allows us more flexibility in specifying properties we wish to check in our models.

Compared to propositional logic, which is concerned with the state of a system at a given point in time, temporal logic allows us to talk about durations and ordering of events.

Topics: Fluent expressions

A fluent is a property that can change over time:

```
fluent FL = <{s1, ..., sN}, {e1, ..., eN}>
```

`FL` is a proposition that is initially false, becomes true when any of the actions in `{s1, ..., sN}` occur, and then becomes false again when any of the actions in `{e1, ..., eN}` occur.

Fluents can be indexed (like processes) and combined using propositional logic connectives (such as `&&, ||, !, ->, <->`).

We can also use universal and existential quantifiers:

```
forall[i:1..2] FL[i]
exists[i:1..2] FL[i]
```

Topics: Temporal logic – always and eventually

Fluents still specify properties at a single point in time, even though their value will depend on their history (trace) up to that point.

The temporal operators always and eventually allow us to specify properties with respect to an entire timeline.

Always: `[]F` is true iff `F` is true at the current instant and at every instant in the future.

Eventually: `<>F` is true iff `F` is true at the current instant or at some instant in the future.

Topics: Temporal logic – until

Always and evenutually are monadic: each is applied to a single formula.

The until operator allows us to specify that a certain property is true until another property becomes true:

`F U G` is true iff G eventually becomes true and `F` is true until that instant.

Topics: Complex systems

- Dynamical systems
- Differential equation-based models
- Cellular automata models
- Agent-based models
- Complex networks (less emphasis on this in 2019)

Models: logistic map, predator-prey, disease spread, 1D CA, game of life, flocking, ant colony foraging, etc.

Preparing for the exam

The exam is no books, no notes, no laptops, no calculators.

In other words, you have to know and understand modelling approaches, concepts, definitions, and notations.

Rely on understanding rather than memorising.

To give you an idea of the format and content of the exam, a practice exam paper is available on the LMS, under "Exam info".

The exam is 3 hours plus 15 minutes reading time.

The published time is the start of reading time—don't miss it.

Beware that the exam venue can be cold!

The exam paper

- The front page's "instructions to students" is available on LMS.
- The exam consists of 11 questions.
- Marks are indicated for each question; be careful to allocate your time accordingly.
- Some questions are harder than others.
- Remember that rationale and justification is important in answering questions *but* also be as concise as possible with your answers.
- Point form is acceptable for more descriptive answers.
- If asked for (eg) two properties/examples/etc. please give two (not five, hoping that two may be correct!) Points *may* be deducted for incorrect information.

Thank you!

Good luck!