Concurrent Programming 19530-V (WS01)

Lecture 4: Interference & Mutual Exclusion

Dr. Richard S. Hall
rickhall@inf.fu-berlin.de

FSP Concepts Review

- A process is a set of ***states*** with ***transitions*** among the various states
  - A process defines all possible/allowable transitions
  - Transitions are equivalent to ***actions***
  - Actions are ***atomic***, which means that they either happen completely or not at all
    - i.e., actions cannot be divided or interrupted
- Processes can be combined to form ***parallel compositions***
  - Parallel compositions are concurrent systems
    - This means that the combined processes conceptually execute all at the same time

FSP Concepts Review

- Process *interaction* occurs when two processes *share* the same action
  - Shared actions are special because they enable processes to *synchronize* with each other
    - A process cannot execute a shared action by itself, a shared action can only execute when all processes that share the action execute it at the same time
  - Shared actions *constrain* a state machine (i.e., they limit the allowable transitions) since they must execute at the same time in all processes
    - Non-shared actions can be arbitrarily interleaved and therefore do not impose constraints
  - Actions can be hidden so that cannot be shared

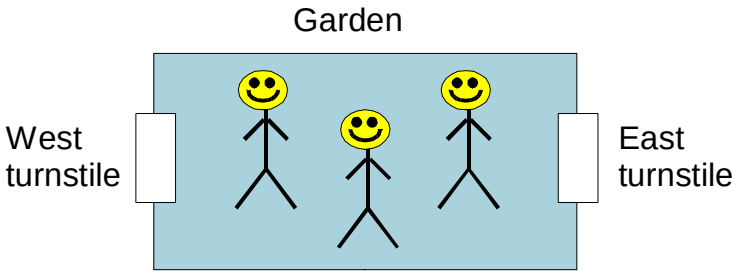
FSP Concepts Review

- Processes can be *re-used* by using a *label prefix* (e.g., **`a:USER`**, **`b:USER`**)
- Labels are also useful when modeling *shared resources*, but in this case we must use a set of prefixes (e.g., **`{a, b}::PRINTER`**)
- In order to define specific process interactions, actions can be *renamed*
- Almost all of these operations are simple textual substitutions, there is no magic

Now we can start to look at the real issues...

Ornamental Garden Problem

People enter an ornamental garden through either of two turnstiles. Management wishes to know how many people are in the garden at any time.



The system model consists of two concurrent processes and a shared counter process.

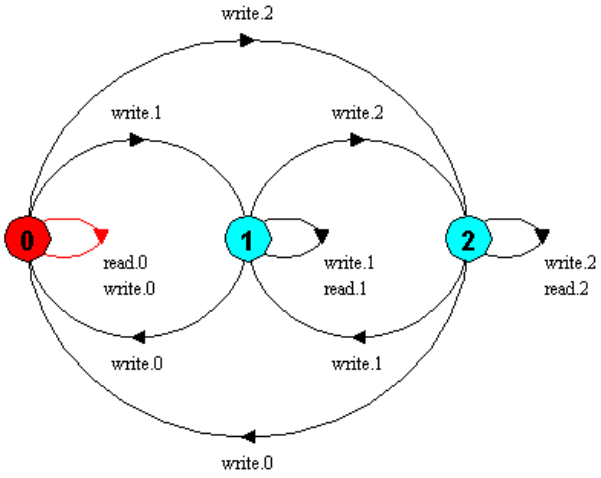
Counter Variable Process

```
VAR      = VAR[0],
VAR[u:T] = (read[u]->VAR[u]
           |write[v:T]->VAR[v]).
```

- How does this process behave?
  - It is initialized to zero
  - It can hold values in the range of **`T`**
  - It allows you to **`read`** a value from it
  - It allows you to **`write`** a value to it

Counter Variable Process

LTS graph for **VAR** process

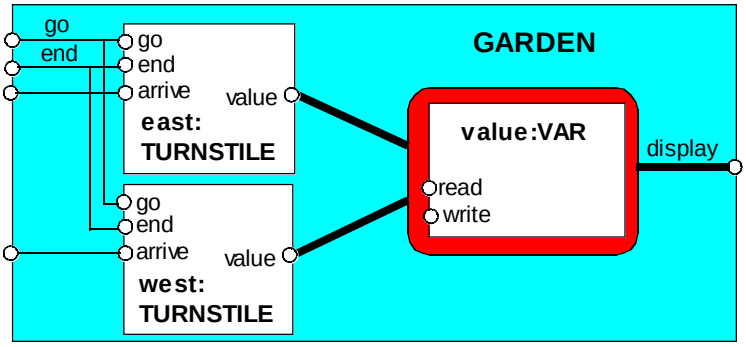


Turnstile Process

```
TURNSTILE = (go->RUN),
RUN       = (arrive->INCREMENT
             |end->TURNSTILE),
INCREMENT = (value.read[x:T]
              ->value.write[x+1]->RUN).
```

- How does this process behave?
  - It is started with **`go`** and accepts **`arrive`**s or can **`end`** at any time
  - Upon an arrival it **`read`**s the value of a counter variable and then **`write`**s a new value and then continues to run

Garden Composition



- The **GARDEN** composition contains
  - Two processes of type **TURNSTILE**, called **east** and **west**
  - One shared process of type **VAR**, called **value**

Garden Composition

The structure diagram helps us to determine how to do relabeling for the composition

```
||GARDEN = (east:TURNSTILE
           ||west:TURNSTILE
           ||{east,west,display}::value:VAR)
            /{go/{east,west}.go,
              end/{east,west}.end}.
```

This is not yet complete, why?

Alphabet Extension

Assume these definitions

```
const N = 4
range T = 0..N
```

What is the alphabet of **VAR**?

```
{ value.read[T], value.write[T] }
```

What is the alphabet of **TURNSTILE**?

```
{ go, arrive, end,
  value.read[T], value.write[1..N] }
```

This causes a problem, why?

Alphabet Extension

- Remember that shared actions constrain the allowable transitions in a FSP
- Action **`write[0]`** is unconstrained since it is not shared with any other process
  - The means that **`write[0]`** can happen at any time
  - This is clearly not a good thing since it would reset the variable back to zero whenever it occurred
- We need some way to constrain **`write[0]`**
  - This is where alphabet extension is useful

Alphabet Extension

Process alphabets are extended by adding actions to it

```
set VarAlpha { value.{read[T], write[T]} }

TURNSTILE = (go->RUN),
RUN       = (arrive->INCREMENT
            |end->TURNSTILE),
INCREMENT = (value.read[x:T]
             ->value.write[x+1]->RUN)
             +VarAlpha.
```

- Alphabet extension adds an action to a process' alphabet, even if the process never performs the action
- The added action, if shared with other process, constrain the state machine like normal shared actions

Complete Garden Example

```
const N = 4
range T = 0..N
set VarAlpha = { value.{read[T],write[T]} }

VAR      = VAR[0],
VAR[u:T] = (read[u]->VAR[u]
           |write[v:T]->VAR[v]).

TURNSTILE = (go->RUN),
RUN       = (arrive->INCREMENT
            |end->TURNSTILE),
INCREMENT = (value.read[x:T]
             ->value.write[x+1]->RUN
            )+VarAlpha.

||GARDEN = (east:TURNSTILE
           ||west:TURNSTILE
           ||{east,west,display}::value:VAR)
            /{go/{east,west}.go,
              end/{east,west}.end}.
```

The alphabet of **TURNSTILE** is extended with **VarAlpha** to ensure there are no unintended free actions in **VAR**, i.e., all actions in **VAR** must be controlled by a **TURNSTILE**.

Garden Implementation

Suppose that we actually implemented the **GARDEN** example in Java and we let each turnstile receive 20 arrive actions, but in the end the counter displays 31 instead of 40. It appears as if some increments have been lost.

Why?

Process Interference

- We model concurrency as the arbitrary interleaving of action from multiple processes
- If processes access a shared object, sometimes the state of the shared object can become incorrect due to certain patterns of action interleaving
  - This is known as ***interference***
- The whole point of concurrent programming is dealing with interference

Finding Errors in Models

Exhaustive checking - compose the model with a **TEST** process which sums the arrivals and checks against the display value

```
TEST       = TEST[0],
TEST[v:T]  =
     (when (v<N){east.arrive,west.arrive}->TEST[v+1]
     |end->CHECK[v]),
CHECK[v:T] =
    (display.value.read[u:T]->
       (when (u==v) right->TEST[v]
       |when (u!=v) wrong->ERROR
       )
    )+{display.VarAlpha}.


||TESTGARDEN = (GARDEN || TEST).
```

Like **STOP**, **ERROR** is a predefined FSP local process (state), numbered -1 in the LTS graph.

Finding Errors in Models

Using the LTSA tool, we can run a safety check on the **TESTGARDEN** process and see if LTSA can find the problem in our **GARDEN** process

```
Trace to property violation in TEST:
     go
     east.arrive
     east.value.read.0
     west.arrive
     west.value.read.0
     east.value.write.1
     west.value.write.1
     end
     display.value.read.1
     wrong
```

This error occurs because the increment operation in ***TURNSTILE*** *is not atomic*

Avoiding Interference Errors

- *Mutual exclusion*
  - Mutual exclusion is a high-level process synchronization concept
  - Mutual exclusion means that a shared resource can only be accessed by one process at a time
    - i.e., processes are not given access to a shared resource if any other process currently has access to that resource
  - Mutual exclusion is achieved with *locks*
    - A lock is modeled as a process that allows an **`acquire`** action followed by a **`release`** action

Avoiding Interference with Locks

Create a locking **VAR** for the **GARDEN** process

```
LOCK = (acquire->release->LOCK).
||LOCKVAR = (LOCK || VAR).
set VarAlpha = {value.{read[T],write[T],
                  acquire, release}}
```

Modify **TURNSTILE** to use the lock

```
TURNSTILE = (go->RUN),
RUN       = (arrive->INCREMENT
             |end->TURNSTILE),
INCREMENT = (value.acquire
              ->value.read[x:T]->value.write[x+1]
              ->value.release->RUN)
              +VarAlpha.
```

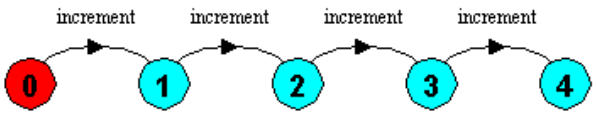
Abstracting Locking Details

```
const N = 4
range T = 0..N

VAR = VAR[0],
VAR[u:T] = (read[u]->VAR[u]
           |write[v:T]->VAR[v]).

LOCK = (acquire->release->LOCK).

INCREMENT = (acquire->read[x:T]
             ->(when (x<N) write[x+1]
                 ->release->increment->INCREMENT
               )
             )+{read[T],write[T]}.

||COUNTER = (INCREMENT||LOCK||VAR)@{increment}.
```

We can hide the locking details of a shared resource by hiding its internal actions and only exposing the desired external actions (e.g., similar to public methods in an object)

Abstracting Locking Details

Minimized LTS for synchronized **COUNTER** process



A simpler process that also describes a synchronized counter

```
COUNTER = COUNTER[0]
COUNTER[v:T] = (when (v<N) increment
                           ->COUNTER[v+1]).
```

*This process generates the same LTS as the previous **COUNTER** definition, thus they describe the same atomic increment behavior*