Shared Objects and Mutual Exclusion

SE 3BB4

Ryszard Janicki

Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada

Shared Objects & Mutual Exclusion

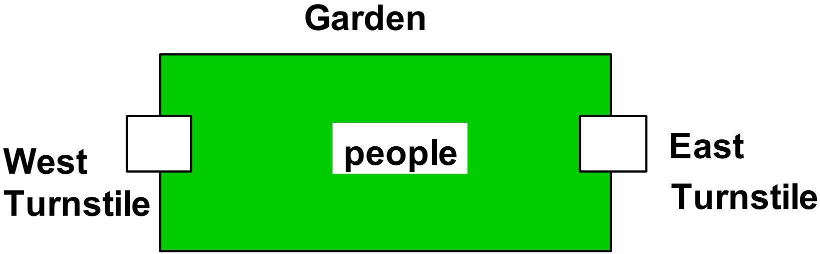
Concepts: process interference.
mutual exclusion and locks.

Models: model checking for interference
modelling mutual exclusion

Practice: thread interference in shared Java objects
mutual exclusion in Java
(synchronized objects/methods).

Ornamental garden problem:

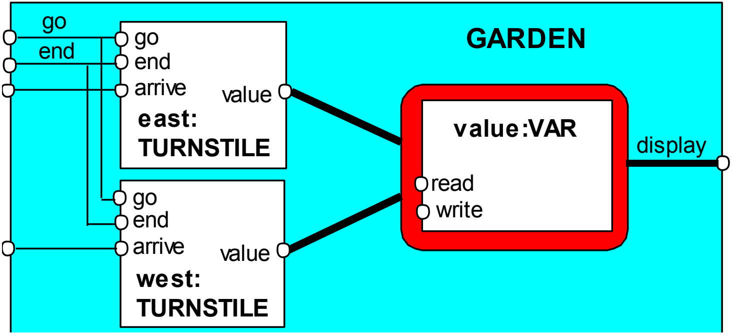
- People enter an ornamental garden through either of two turnstiles. Management wish to know how many are in the garden at any time.



- Suppose that movement of people is modeled by two concurrent processes and a '*shared*' counter.

First Solution to The Garden Problem

- *Simplification*: Nobody leaves the garden
- Technique: Busy Waiting
- We have to implement counting!
- Structure Diagram of Ornamental Garden:



```
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 shared process **VAR** is declared explicitly as a **set** constant, **VarAlpha**.

The **TURNSTILE** alphabet is extended with **VarAlpha** to ensure no unintended free (autonomous) actions in **VAR** such as **value.write[0]**.

All actions in the shared **VAR** must be controlled (shared) by a **TURNSTILE**.

- *go*/{*east*, *west*}.*go* means *east*.*go* and *west*.*go* are the same as action *go*.
- *go*/{*east*, *west*}.*end* means *east*.*end* and *west*.*end* are the same action as *end*.
- but *east*.*arrive* and *west*.*arrive* are distinct!

- $\{east, west, display\} :: value : VAR$ implies:

$$value : VAR = value : VAR[0]$$

$$value : VAR[u : T] = (value.read[u] \rightarrow value : VAR[u] \mid value.write[v : T] \rightarrow value : VAR[c]) =$$

$$(east.value.read[u] \rightarrow value : VAR[u] \mid$$
$$west.value.read[u] \rightarrow value : VAR[u] \mid$$
$$display.value.read[u] \rightarrow value : VAR[u] \mid$$
$$east.value.write[v : T] \rightarrow value : VAR[v] \mid$$
$$west.value.write[v : T] \rightarrow value : VAR[v] \mid$$
$$display.value.write[v : T] \rightarrow value : VAR[v])$$

- Consider a trace:

$go \rightarrow east.arrive \rightarrow east.value.read[0] \rightarrow west.arrive \rightarrow west.value.read[0] \rightarrow east.value.write[1] \rightarrow west.value.write[1] \rightarrow end \rightarrow display.value.read[1]$

- We have **two** people in the garden but the counter displays number 1!
- $west.value.read[0]$ was executed *before* $east.value.write[1]$, so *VAR* did not update storage!
- The trace below is OK.

$go \rightarrow east.arrive \rightarrow east.value.read[0] \rightarrow east.value.write[1] \rightarrow west.arrive \rightarrow west.value.read[1] \rightarrow west.value.write[2] \rightarrow end \rightarrow display.value.read[2]$

- How can we find such errors?
- Exhoustive checking, a kind of model checking, software support needed
- **TEST:** a process which summs 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}.
```

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

$\| \, TESTGARDEN = (GARDEN \, \| \, TEST)$

- LTSA will produce the red trace form page 7 followed by '*wrong*'

Interference and Mutual Exclusion

- Destructive update, caused by the arbitrary interleaving of *read* and *write* type actions, is termed **INTERFERENCE**.
- Interference bugs are extremely difficult to locate.
- The general solution is to use **MUTUAL EXCLUSION**.

Modeling Mutual Exclusion

- To add locking to our model, define a LOCK, compose it with the shared VAR in the garden, and modify the alphabet set :

```
LOCK = (acquire->release->LOCK).
||LOCKVAR = (LOCK || VAR).

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

- Modify TURNSTILE to acquire and release 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.
```

- Old INCREMENT:
  $INCREMENT = (value.read[x : T] \rightarrow value.write[x + 1] \rightarrow RUN)$
  $+VarAlpha$

- Trace:

$go \rightarrow east.arrive \rightarrow east.value.acquire \rightarrow east.value.read[0] \rightarrow east.value.write[1] \rightarrow east.value.release \rightarrow west.arrive \rightarrow west.value.acquire \rightarrow west.value.read[1] \rightarrow west.value.write[2] \rightarrow west.value.release \rightarrow end \rightarrow display.value.read[2]$.

- We can test it similarly as previously using TEST process and LTSA.
- But tests cannot prove correctness, only can find errors!

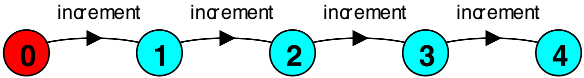
Abstraction using action hiding

```
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}.
```

To model shared objects directly in terms of their synchronized methods, we can abstract the details by hiding.

For **SynchronizedCounter** we hide **read, write, acquire**, **release** actions.

LTS:



- Another simpler COUNTER (the same LTS):

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

- $\| \; COUNTER \approx COUNTER$,
  old and new counters are *bisimilar*, i.e. equivalent.

Another Model

- Is the model discussed really necessary?
- Can it be specified in a much more simpler way?
- What about this:

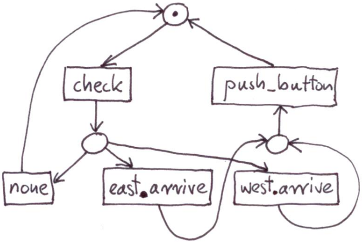
$TURNSTILES\_GUARD = (check \rightarrow$
$(east.arrive \rightarrow push\_button \rightarrow TURNSTILE\_GUARD \mid$
$west.arrive \rightarrow push\_button \rightarrow TURNSTILE\_GUARD \mid$
$none \rightarrow TURNSTILE\_GUARD)$

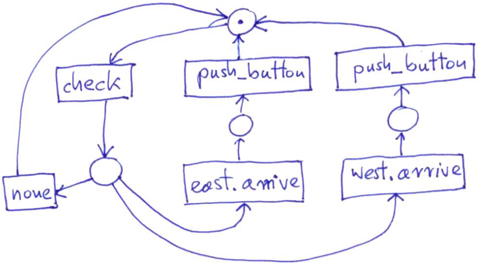
$COUNTER\_TO\_4 = push\_button \rightarrow +1 \rightarrow push\_button \rightarrow +1 \rightarrow$
$push\_button \rightarrow +1 \rightarrow push\_button \rightarrow +1 \rightarrow STOP$

$\| \; GARDEN = (TURNSTILES\_GUARD \, \| \, COUNTER\_TO\_4)$

Petri Nets Versions

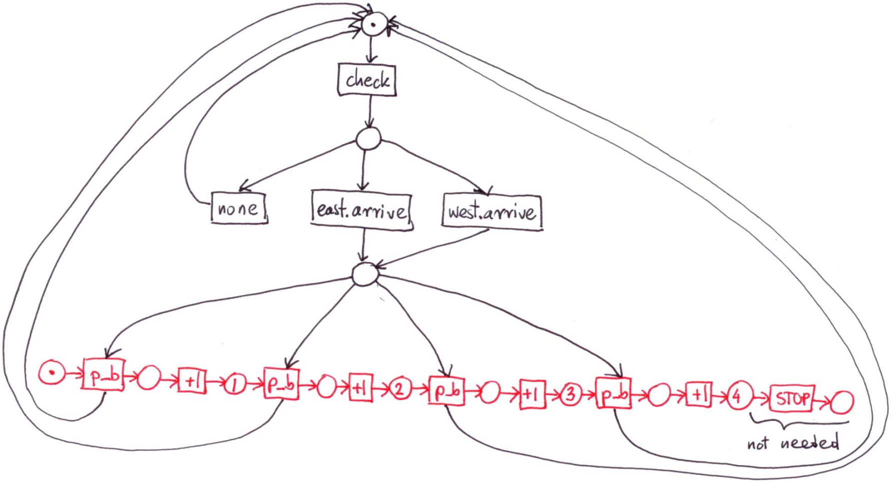
TURNSTILES_GUARD (two versions):





COUNTER_TO_4:

Composed Net



Simultaneity

- Suppose that the garden we have considered is a sacred garden of some cult that worship '*simultaneity*'.
  Hence it has two counters, one that counts all worshipers in the garden, and the other that only counts those blessed events when two people enter simultaneously from both the east and the west entrances.
- Assume that this event is somehow observable, for instance openings og gates are synchronized, etc.
- Can you model this new garden in terms of FSP?
  If 'yes', how, as we have interleavings only in this model?