SWEN90004
Modelling Complex Software Systems

Synchronisation in FSP

Nic Geard

Lecture Con.07

Semester 1, 2019

Interference and related problems

We have seen how to create threads in Java, and looked at some of the problems that threads with shared data can create (compared with sequential programs). We also looked at modelling concurrent processes in FSP.

Now we look at how we can model problems in FSP and check for properties such as deadlock and interference. Doing this allows us to model concurrent systems at a level that will give us a greater chance of identifying potential problems.

We can use LTSA to search for these problems automatically.

Deadlock

A process is in a deadlock if it is blocked waiting for a condition that will never become true.

A process is in a livelock (a busy wait deadlock) if it is spinning while waiting for a condition that will never become true. Either can happen if concurrent processes or threads are mutually waiting for each other.

The Coffman conditions

Coffman, Elphick, and Shoshani identify four necessary and sufficient conditions (the Coffman conditions) that all must occur for deadlock to happen:

1. Serially reusable resources: the processes involved must share some reusable resources between themselves under mutual exclusion.
2. Incremental acquisition: processes hold on to resources that have been allocated to them while waiting for additional resources.
3. No preemption: once a process has acquired a resource, it can only release it voluntarily—it cannot be forced to release it.
4. Wait-for cycle: a cycle exists in which each process holds a resource which its successor in the cycle is waiting for.

Bounded buffers using monitors

Let us consider an scenario in which deadlock is a possibility. This example uses monitors and is discussed in a workshop: the bounded buffer.

The buffer consists of a number of fixed slots. Items can be put into the buffer by a producer process, and taken from the buffer by a consumer process in a first-in first-out (FIFO) manner.

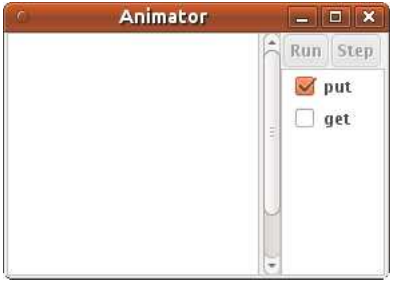
An item can only be put into the buffer if there is a free slot; otherwise the calling producer is blocked. An item can only be removed from the buffer if there is such an item in the buffer; otherwise the calling consumer is blocked.

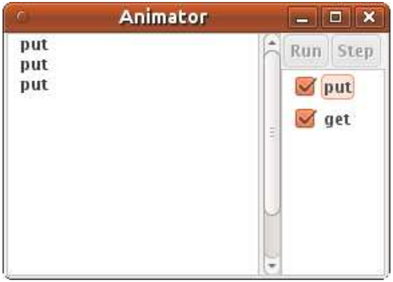
The bounded buffer in FSP

```
BUFFER(N=5) = COUNT[0],
COUNT[i:0..N]
  = ( when (i<N) put -> COUNT[i+1]
    | when (i>0) get -> COUNT[i-1]
    ).

PRODUCER = (put -> PRODUCER).
CONSUMER = (get -> CONSUMER).

||BOUNDEDBUFFER = (PRODUCER || BUFFER(5) || CONSUMER).
```

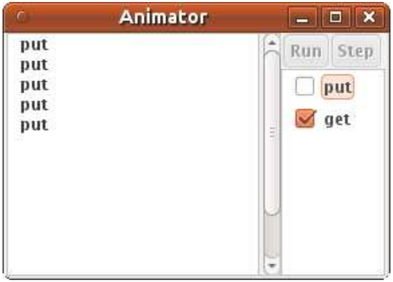
There is no consideration of the items in the buffer at all. The model only includes whatever is relevant to the interaction between processes. Keeping track of the number of buffered items is sufficient for this, and is preferred, as it abstracts away details that are irrelevant from a concurrency point of view.

Animating the bounded buffer model in LTSA





```
put
put
put
```



```
put
put
put
put
put
```

The screen shots correspond to an empty buffer (only put is enabled), a half-full buffer (both put and get are enabled), and a full buffer (only get is enabled).

FSP vs Java

FSP monitors map well to Java monitors. In particular, the design template for waiting in Java monitors can be mapped directly from FSP guarded processes, such as below.

```
when cond act -> NEWSTAT
```

becomes

```
public synchronized void act()
    throws InterruptedException
{
  while (!cond) wait();
  //modify monitor data
  notifyAll();
}
```

FSP vs Java

The difference in the level of abstraction between FSP and Java means that `cond` will not always be exactly the same. For example, consider the bounded buffer example from Workshop Con.02.

In the implementation of `get()`, the condition is

```
while (buffer.size() == 0)
    wait();
```

The size of the buffer is equivalent to `i` in the FSP model.

Modelling semaphores in FSP

Instead of monitors, let us try to use semaphores to synchronise the uses of the bounded buffer (as in Lecture Con.04).

Let us use the shorter “up” and “down” for “signal” and “wait”.

We use “empty” for the semaphore that blocks when the buffer is empty, and “full” for the one that blocks when the buffer is full.

The bounded buffer using semaphores in FSP

Each semaphore will block when its value is 0, so the `empty` semaphore is initialised to `N` (we can "put" something in the buffer `N` times without an intervening "get"); similarly the `full` semaphore is initialised to 0. (So the `full` semaphore will block calls to `get` initially.)

Given a `put`, the `empty` semaphore is decremented, and the `full` semaphore is incremented. Given a `get`, `full` is decremented and `empty` is incremented.

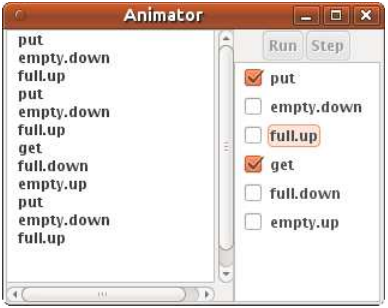
A model for this is shown on the following slide.

The bounded buffer using semaphores in FSP

```
const N = 5
range Int = 0..N

SEMAPHORE(I=0) = SEMA[I],
SEMA[v:Int]    = ( when (v<N) up   -> SEMA[v+1]
                 | when (v>0) down -> SEMA[v-1]
                 ).

BUFFER    = ( put -> empty.down -> full.up -> BUFFER
            | get -> full.down -> empty.up -> BUFFER
            ).
PRODUCER = (put -> PRODUCER).
CONSUMER = (get -> CONSUMER).

||BOUNDEDBUFFER = (  PRODUCER || BUFFER || CONSUMER
                  || empty:SEMAPHORE(N)
                  || full:SEMAPHORE(0)
                  ).
```

The bounded buffer using semaphores in FSP

But is the model correct? To investigate, use the LTSA animator.



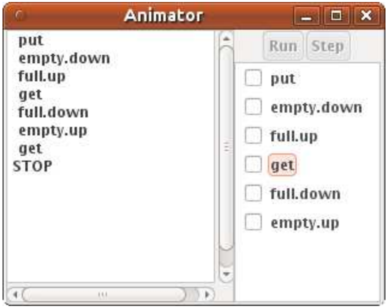
```
put
empty.down
full.up
put
empty.down
full.up
get
full.down
empty.up
put
empty.down
full.up
```

- [✓] put
- [ ] empty.down
- [ ] full.up
- [✓] get
- [ ] full.down
- [ ] empty.up

The trace shows that we can put and get items into the buffer. We can animate many more traces to obtain more confidence that our model is correct.

Deadlock with the bounded buffer

But this trace shows that the bounded buffer can deadlock:



```
 put
 empty.down
 full.up
 get
 full.down
 empty.up
 get
STOP
```

The `get` transition is enabled when the buffer is empty. A request to `get` will suspend until the `full` semaphore can be decremented, which is impossible, as it is 0. And the `put` action is now disabled, since `get` has been executed, locking the buffer monitor.

Deadlock with the bounded buffer

Now the process is in a deadlocked state in which the consumer is waiting for something to be put into the buffer, but the producer cannot put anything in.

The deadlock can be identified by seeing that the `STOP` process has occurred (and `STOP` is not even present in our model), and by also noting that no actions are enabled.

In this case, we got lucky and managed to run a trace to deadlock. However, it is easy to see that for real models, we may not get so lucky.

Like software testing, this is downside of animation: it can only show the presence of faults, never their absence.

Checking the semaphore example

However, we can check for deadlock automatically using LTSA.

In LTSA, model checking performs a complete breadth-first search on the corresponding LTS, terminating when either:

1. it finds a state with no outgoing transitions (that is, a deadlock has occurred); or
2. it has searched all states (no deadlock).

When a deadlock is found, the breadth-first approach has found a shortest possible trace to deadlock.

Checking the semaphore example

To find deadlocks in LTSA, the check that we will use is a safety check.

The default safety check is a check for deadlock. To check for deadlock, select *Check* $\rightarrow$ *Safety* from the menu.

If we do this for the bounded buffer example, we get the following:

```
Trace to DEADLOCK:
    get
Analysed in: 0ms
```

Checking the semaphore example

This is consistent with our animation that shows executing the get action when the buffer is empty results in a deadlock.

This is far better than using the animator, because we know that if a deadlock exists in the model, we will always find a trace to the deadlock—the model checking part of LTSA will search all possible states.

Even better, we will have the shortest possible trace to it.

The problem with the semaphore solution

The situation described in the bounded-buffer-with-semaphore example is known as the nested monitor problem. It occurs because the four Coffman conditions all apply.

The reason why it occurs with the semaphore-guarded buffer and not with the original buffer is that the semaphore solution introduces incremental acquisition—the 2nd Coffman condition. By executing `get`, the process obtains the lock for the buffer, and then tries to claim the `full` semaphore as well.

Correcting the bounded buffer behaviour

To fix the problem, we re-design the buffer so that the buffer lock is not acquired until after the semaphores have been acquired:

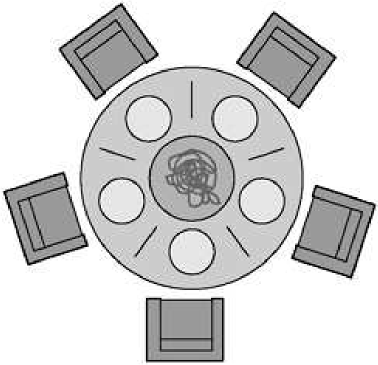
```
BUFFER = ( empty.down -> put -> full.up -> BUFFER
         | full.down -> get -> empty.up -> BUFFER
         ).
```

This removes the deadlock. However, given a half-full buffer, if either semaphore is acquired, the other process will be blocked from the other semaphore. A more efficient design is to leave the semaphore access to the producer and consumer:

```
BUFFER   = (put -> BUFFER | get -> BUFFER).
PRODUCER = (empty.down -> put -> full.up -> PRODUCER).
CONSUMER = (full.down -> get -> empty.up -> CONSUMER).
```

The dining philosophers problem

Five philosophers share a circular table. Each spends his/her life alternately thinking and eating. In the centre of the table is a large plate of spaghetti. A philosopher needs two forks to eat a helping of spaghetti. Unfortunately, as philosophy is not as well paid as computing, the philosophers can only afford five forks. One fork is placed between each pair, and they agree that each will only use the forks to their immediate right and left.



Dining philosophers in FSP

Each fork is a shared resource. It can be picked up then put down, repeatedly:

```
FORK = (get -> put -> FORK).
```

A philosopher picks up two forks, one at a time. He/she sits down, gets both forks, eats, puts the forks down, and finally stands up, ready to start thinking again:

```
PHIL = (sitdown -> right.get -> left.get -> eat
        -> left.put -> right.put -> arise -> PHIL).
```

Dining philosophers in FSP

```
FORK = (get -> put -> FORK).
PHIL = (sitdown -> right.get -> left.get -> eat
        -> left.put -> right.put -> arise -> PHIL).
```

Finally, to put the five philosophers together with the fork resources, we use the following composite process:

```
||DINERS(N=5) =
   forall [i:0..N-1]
     (   phil[i]:PHIL
     ||  {phil[i].left,phil[((i-1)+N)%N].right}::FORK
     ).
```

Note that `((i-1)+N)%N` is just decrement modulo `N`.

Deadlocking philosophers

Using the LTSA safety check, the following deadlock is found:

The deadlock is caused by all philosophers sitting down together, and each getting the fork to their right.

Clearly, the fourth Coffman condition occurs: a wait-for cycle.

To obtain a deadlock-free system, we must alter one of the four conditions.

We will remove the wait-for cycle.

```
Trace to DEADLOCK:
    phil.0.sitdown
    phil.0.right.get
    phil.1.sitdown
    phil.1.right.get
    phil.2.sitdown
    phil.2.right.get
    phil.3.sitdown
    phil.3.right.get
    phil.4.sitdown
    phil.4.right.get
```

Deadlock-free philosophers

In the previous version of the dining philosophers, each philosopher could pick up their right fork at the same time, ending in deadlock due to the wait-for cycle. There are no general methods for removing wait-for cycles. We just have to think carefully about our designs to ensure they do not exist.

Having FSP and the LTSA tool set helps, as we can assess different designs and prove them free of deadlock before implementation.

One way to get around the problem of all philosophers behaving the same way at the same time is to have them behave differently. In the previous version, the philosophers all had the same definition.

Deadlock-free philosophers

To remove the deadlock, let odd-numbered philosophers pick up their right fork first, and even-numbered ones pick their left fork first:

```
PHIL(I=0)
  = ( when (I%2 == 0)
        sitdown -> left.get -> right.get
        -> eat -> left.put -> right.put
        -> arise -> PHIL
    | when (I%2 == 1)
        sitdown -> right.get -> left.get
        -> eat -> left.put -> right.put
        -> arise -> PHIL
    ).
```

The LTSA safety check confirms this solution is deadlock free.

References / Recommended reading

- E. Coffman, M. Elphick, and A. Shoshani: System deadlocks, *ACM Computing Surveys* 3(2): 67–78, 1971.
- J. Magee and J. Kramer, *Concurrency: State Models and Java Programs*, 2nd edition, John Wiley and Sons, 2006. Available at `http://flylib.com/books/en/2.752.1.1/1/`