SWEN90004
Modelling Complex Software Systems

Checking safety in FSP

Nic Geard

Lecture Con.08

Semester 1, 2019

Interference and mutual exclusion

Back in Lecture Con.01 we studied two threads that were incrementing the same variable.

The load-and-store semantics meant that one thread could read and increment the variable, and then the second thread could come in and read the old value before the first thread had written the new value back.

The FSP model on the next slide captures this.

Interference and mutual exclusion

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

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

CTR = ( read[x:T] ->
        ( when (x<N) increment -> write[x+1] -> CTR
        | when (x==N) end -> END
        )
      )+{read[T],write[T]}.

||SHAREDCOUNTER = ({a,b}:CTR || {a,b}::VAR).
```

At the end of the CTR process, we see some new FSP syntax called an alphabet extension: `+{read[T],write[T]}`.

FSP—Alphabet extensions

Recall that the alphabet of a process is the set of actions in which it engages. The `CTR` process has the alphabet

$$\{\texttt{read[0]}, \ldots, \texttt{read[4]}, \texttt{write[1]}, \ldots, \texttt{write[4]}\}.$$

Note that the action `write[0]` is not part of the alphabet because the process never engages in it.

On its own, this is fine. But when we compose `CTR` with `VAR`, this means the action `write[0]` (from `VAR`) is free to execute at any time. Extending the alphabet of `CTR` prevents this problem.

Exercise:
Remove the alphabet extension from `CTR` and compare the difference between the old and new LTS for `SHAREDCOUNTER`. (Do it for `N = 2`.)

Animating the *Counter* example

Below are two traces: one that finds the error (on the left), and one that does not (on the right).

Finding the problem depends on luck.

Checking the *Counter* example

If we use the LTSA safety check, we see this:

```
No deadlocks/errors
Analysed in: 1ms
```

This is fine, but we want to know whether there is interference. One way to detect this is to find a trace such that both processes write the same value. To do this, we extend the model:

```
INTERFERENCE = (a.write[v:T] -> b.write[v] -> ERROR).
||SHAREDCOUNTER
  = ({a,b}:CTR || {a,b}::VAR || INTERFERENCE).
```

The process `ERROR` is a pre-defined process that signals an error in the model, and causes a deadlock.

Checking the *Counter* example

So now the `SHAREDCOUNTER` process only allows traces in which a sequence `a.write[v:T] -> b.write[v]` does not occur. If it does occur, the process terminates in error.

If we use the safety check in LTSA to search for deadlocks, then by finding a deadlocked trace that executes `ERROR`, we will know that both processes have written the same variable. Running the extended model, we get the output:

```
Trace to property violation in INTERFERENCE:
    a.read.0
    a.increment
    b.read.0
    a.write.1
    b.increment
    b.write.1
```

Checking the *Counter* example

```
Trace to property violation in INTERFERENCE:
    a.read.0
    a.increment
    b.read.0
    a.write.1
    b.increment
    b.write.1
```

This provides us with a trace that confirms that both processes are writing the value 1, that is, we have interference.

Again, this is better than using the animator, because we know that we will always find such a trace—the model checking function of LTSA will search all possible states.

Mutual exclusion

To prevent the interference, we need a solution similar to the solutions seen in earlier lectures for mutual exclusion. One advantage of working with FSP instead of Java is that we can disregard all irrelevant details.

First let us create a "lock" process, which allows synchronisation between the `CTR` processes:

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

Then we modify `CTR` so that it has to acquire a lock before executing the critical section, and release it afterwards.

Mutual exclusion

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

Finally, we use the lock:

```
||LOCKED_SHAREDCOUNTER
  = ({a,b}:CTR || {a,b}::(LOCK||VAR)).
```

Mutual exclusion

If we run a safety check on this example, we get no deadlocks.

However, we still need to test for interference.

If we add the `INTERFERENCE` process again, we get the following:

```
Trace to DEADLOCK:
    b.acquire
    b.read.0
    b.increment
```

This is good, in that it failed to find a trace for interference. But it does not mean there is no such trace. LTSA just found some other trace first, which is deadlocked (since `INTERFERENCE` happened to expect a to write before `b`).

Mutual exclusion

If we analyse the LTS (which is too large for these slides), we see that the reason we reach a deadlock because LTSA cannot generate an LTS that contains a trace in which both processes write the same value.

This means that our system is free from interference.

We could try to make `INTERFERENCE` far more sophisticated so that it always reports no deadlocks when a trace does not occur.

However, as it turns out, it is not necessary to go to that trouble.

Checking safety and liveness properties

We will be looking at how to check properties of FSP models using LTSA. The basic methodology is that the user describes some concurrent processes using FSP, and then describes a property of that model. A property is some attribute of a model that is true for every possible trace/execution of that model.

In concurrent systems, there are two categories of property that are of interest:

- Safety properties: A safety property asserts that nothing "bad" happens during execution. A deadlock is an example of this.
- Liveness properties: A liveness property asserts that some "good" eventually happens. For example, that all processes trying to access a critical section eventually do get access.

Safety and liveness properties

In sequential systems, the most common safety property is that the system satisfies some assertion each time a given program point is reached.

In concurrent systems, we have seen that additional important safety properties are absence of deadlock and interference.

The most common liveness property for a sequential system is that the system terminates.

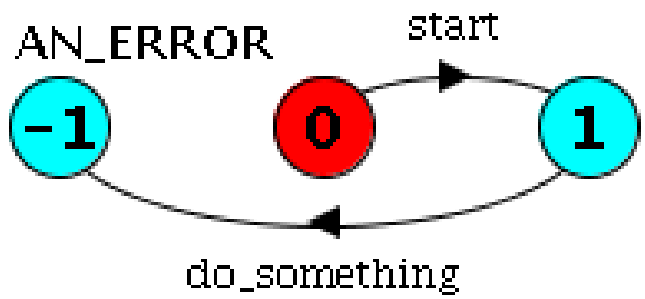
Concurrent systems are often designed to be non-terminating, and liveness properties are most commonly related to resource access.

Error states in FSP and LTS

Earlier we used the special pre-defined process `ERROR`. This process signals termination in an error state, that is, a state that we should never want to move into. Here is a simple process that ends in error:

```
AN_ERROR = (start -> do_something -> ERROR).
```

The LTS for this is:



Note the state labelled -1. This is a special state ID that indicates the `ERROR` process. It has no outgoing transitions.

Error states in FSP and LTS

If we run the LTSA safety check, we get the following violation:

```
Trace to property violation in AN_ERROR:
    start
    do_something
```

Modelling with error states

The `ERROR` process is used in FSP models to indicate erroneous behaviour. This way we can explicitly identify erroneous actions for the people who will implement the model.

Consider this example of an actuator responding to commands. It must respond to a command before receiving the next command:

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

Modelling with error states

Running the LTSA safety check on this results in a violation:

```
Trace to property violation in ACTUATOR:
    command
    command
```

But the downside to this approach is also clear.

We have a trace that shows a property violation, despite the fact the model arguably behaves as intended.

In particular, if we had a deadlock elsewhere in the system, that might not be detected, because LTSA shows only the first deadlock it encounters.

Safety properties

When modelling complex systems, it is better practice to consider only the desired system behaviour, rather than also try to enumerate all possible undesirable behaviours. So, given a model, specify some desirable properties and check that the model maintains them.

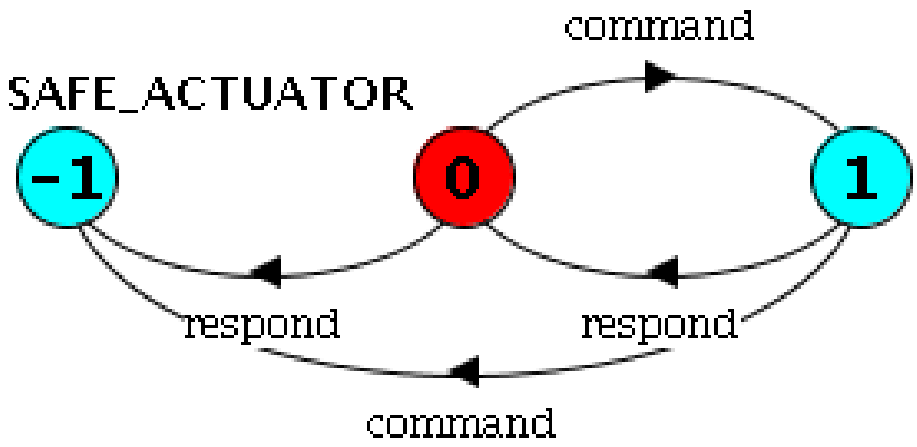
In many formalisms, safety properties are used for this purpose. In FSP, a safety property is just a process, but to identify it clearly as a safety property, FSP uses the `property` keyword:

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

This property says that, whenever a `command` action is observed, a `respond` action should occur before another `command` action occurs.

Safety properties and error states

The LTS for the SAFE_ACTUATOR property is this:



Note that the error states are automatically generated. This LTS describes a process that goes to an error state whenever an action occurs out of sequence. For example, two command actions occurring in a row is an error.

Safety properties and error states

The LTSA compiler generates the LTS as if `SAFE_ACTUATOR` is a normal FSP process, and then, for every state in the LTS, it adds an outgoing action for all actions in the process's alphabet that are not already outgoing actions. These new outgoing actions go to the error state. As a result, the LTS is complete: all actions can occur from all states. Invalid actions go to the error state. Therefore, every possible combination of actions are permitted.

To maintain this transparency, safety properties must be deterministic processes. That is, they must not contain non-deterministic choices.

Using safety properties: Example 1

We modify the earlier ACTUATOR process to remove the error state, but leave in the error that was made (two command actions in row).

Then we compose this with the SAFE_ACTUATOR property:

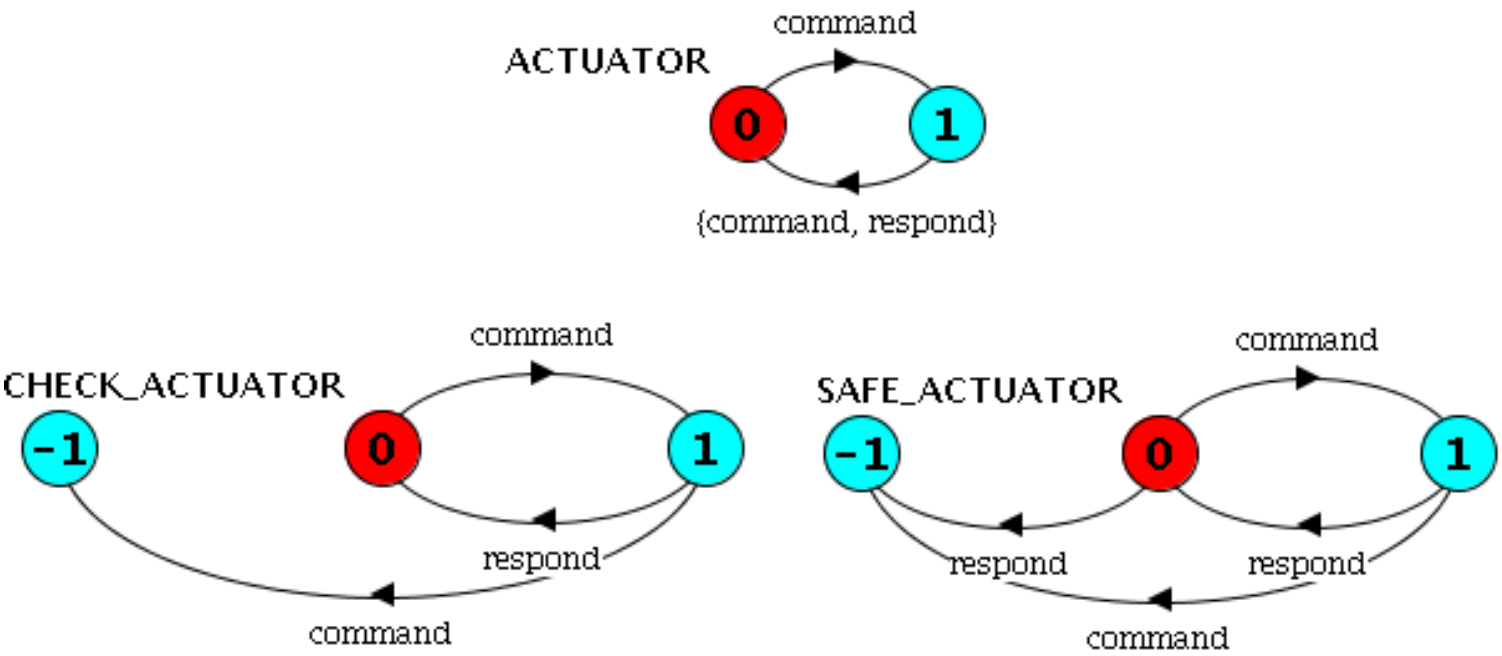
```
ACTUATOR = (command -> ACT),
ACT = (respond -> ACTUATOR | command -> ACTUATOR).

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

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

Using safety properties: Example 1

Consider the LTSs generated for these:



Using safety properties

An LTS generated from a property process allows every possible combination of actions in a process's alphabet. As a result, composing a property process with a normal process, as we have done with `ACTUATOR` and `SAFE_ACTUATOR`, means that we do not affect the normal behaviour of the original process: all previous transitions remain because all shared actions can be synchronised. If behaviour that violates the safety property occurs, the result in the composite process will be the error state.

In our example, the property process `SAFE_ACTUATOR` accepts any action from any state (except the error state). However, `CHECK_ACTUATOR` does not accept the sequence `respond -> respond`, because, when `SAFE_ACTUATOR` is composed with `ACTUATOR`, `respond` is a shared action. As the composite cannot synchronise these, there is no error trace `respond -> respond`.

Using safety properties

Note that the sequence `command -> command` is permitted by `CHECK_ACTUATOR`, because both simple processes allow it. However, because the safety property does not specify it as valid behaviour, this sequence ends at the error state. Using the safety check in LTS, we get:

```
Trace to property violation in SAFE_ACTUATOR:
    command
    command
```

Example 2: A safety property for interference

Earlier we used LTSA to find a case of interference.

In that example, when we had an interference-free model, that was difficult to verify because the `INTERFERENCE` process, when composed with the `SHAREDCOUNTER` process, led to the following deadlock:

```
Trace to DEADLOCK:
    b.acquire
    b.read.0
    b.increment
```

Using a safety property avoids this pitfall, because a safety property's LTS is complete.

Example 2: A safety property for interference

The safety property for interference is that each value is written only once. To capture this, let us use a stronger safety property: when a value `v` is written, the next value written is `v+1`:

```
property NO_INTERFERENCE
  = ({a,b}.write[v:T] ->
      ( when (v<N)
          {a,b}.write[v+1] -> NO_INTERFERENCE
      )
    ).
```

Recall that `{a,b}.write` means that either `a` or `b` can engage in the `write` action. Hence this safety property is agnostic about who writes the value, as long as the next value is one more. The guard `when (v<N)` prevents the value `N+1` from being written.

Example 2: A safety property for interference

For the version of the system that did not use a lock, running this through the LTSA safety check has the same result as before:

```
Trace to property violation in NO_INTERFERENCE:
    a.read.0
    a.increment
    b.read.0
    a.write.1
    b.increment
    b.write.1
```

Example 2: A safety property for interference

However, when we update the model to include a lock, we get:

```
No deadlocks/errors
```

This gives us much more confidence that there is no interference in our model than when we used the `INTERFERENCE` process.

Example 3: A safety property for mutual exclusion

As another example, we'll look again at mutual exclusion.

We will use the `SEMAPHORE(N)` process defined in the Lecture Con.07.

In this example, let us model `M` concurrent loops, each possibly requiring access to a critical section. Each loop executes the following, where `mutex` is the name of a semaphore:

```
LOOP
 = (mutex.down -> enter -> exit -> mutex.up -> LOOP).
```

The abstraction here is key: we only want to model the parts relevant to the interaction between processes, which include the semaphore and entering/exiting the critical section.

Example 3: A safety property for mutual exclusion

A model of `M` loops is:

```
||M_LOOPS
  = (   p[1..M]:LOOP
    ||  {p[1..M]}::mutex:SEMAPHORE(1)
    ).
```

Here the `M` loops are composed with a binary semaphore.

That is, `SEMAPHORE(1)` will block after one `down` action because there is only one resource: the critical section.

Example 3: A safety property for mutual exclusion

The safety property for mutual exclusion is straightforward: when a process enters the critical section, the same process must exit the critical section before another can enter:

```
property MUTEX
  = (p[i:1..M].enter -> p[i].exit -> MUTEX).
```

If we check this using LTSA, we find that there are no violations of the safety property.

Example 3: A safety property for mutual exclusion

In contrast, consider a case where we did not use a binary semaphore, but instead used the process `SEMAPHORE(2)`. This will allow two processes in the critical section at a time.

Model checking this using LTSA reveals the violation:

```
Trace to property violation in MUTEX:
    p.1.mutex.down
    p.1.enter
    p.2.mutex.down
    p.2.enter
```

This is as we would want: it demonstrates that it is possible that two processes enter the critical section at the same time.