SWEN90004
Modelling Complex Software Systems

Checking liveness in FSP

Nic Geard

Lecture Con.09

Semester 1, 2019

Introduction

In the last lecture we studied the use of LTSA for checking safety properties.

Now we turn to liveness.

Liveness properties

A liveness property can be considered the opposite of a safety property: something "good" happens eventually.

We shall discuss one type of liveness property: progress. Progress properties are used to state that a specified action will eventually execute. This is the opposite of starvation, discussed in Lecture Con.02.

Progress properties are simpler to specify than safety properties, yet they are still powerful. Starvation can be as harmful to a system as deadlock if the starved process is at all critical.

To introduce progress, let us consider the tossing of a coin:

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

Fair choice

If we tossed a coin an infinite number of times, we would see heads infinitely often, and tails infinitely often. However, this assumes that the coin is fair. Similarly, if we implement the `COIN` model, we would only see both heads and tails if the implemented process was fair. We could easily implement an unfair version that always resulted in `heads`, which would satisfy the model, but probably not users.

Magee and Kramer define fair choice:

> *"If a choice over a set of transitions is executed infinitely often, then every transition in the set will be executed infinitely often."*

If a single transition occurs infinitely often in our system, then it must be the case that, at any state, that action will occur at some point in the future.

FSP – Progress properties

We assert progress in FSP using a progress property.

> "`progress P = {a1, a2, ..., aN}` *defines a progress property* `P` *that asserts that, in an infinite execution of a target system, at least one of the actions* `a1, a2, ..., aN` *will be executed infinitely often.*"

For the `COIN` process, we assert two liveness properties:

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

Now check the properties by selecting *Check* → *Progress* in LTSA:

```
No progress violations detected.
```

A trickster's coin

The coin example demonstrated a case in which progress was achieved. Let us consider a modified coin example where progress is not achieved. In this example, a trickster can choose between a one-sided coin that always comes up heads, or a normal coin:

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

A trickster's coin

Checking the same progress properties in LTSA uncovers this:

```
Progress violation: TAILS
Trace to terminal set of states:
    pick
Cycle in terminal set:
    toss
    heads
Actions in terminal set:
    {heads, toss}
```

We were right to be suspicious: tails may never occur.

In the above error message, the first line reveals the violation (the `TAILS` progress property); the rest provides debugging information that we decode shortly.

A trickster's coin

Note that the following progress property is not violated by the TWOCOIN process:

```
progress HEADSTAILS = {heads, tails}
```

This property holds because only one of the actions in the set needs to occur infinitely often. In the TWOCOIN process, the heads action will occur infinitely often because it occurs in both the normal coin and the trickster's coin.

Progress analysis in LTSA: SCCs

To help decode the output, and therefore help us debug our models, we'll look at what LTSA does to check progress properties.

The first step is to find all of the terminal sets of states in the model.

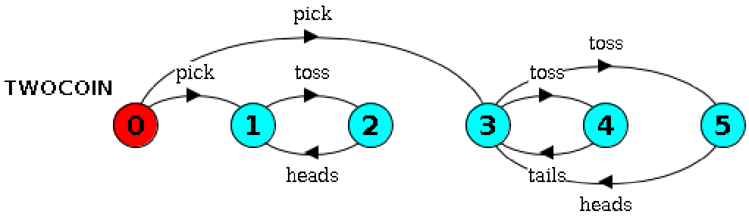
A terminal set of states is a strongly connected component (or SCC) of the LTS.

The SCCs are the equivalence classes of nodes under the "are mutually reachable" relation.

In other words, a terminal set T of states is a set in which every state in T is reachable from every other state in T, and there is no transition from within T to a state outside T.

Progress analysis in LTSA: SCCs

As an example, consider the LTS for the `TWOCOIN` process:



$\{1, 2\}$ is a strongly connected component, and so is $\{3, 4, 5\}$.

Progress analysis in LTSA

The second step is to find the actions that can happen only finitely often.

Recall that FSP models have a finite number of states. For a state to be visited infinitely often, it must be in a terminal set. As we assume fair choice, unless an action is used in every terminal set, it cannot be guaranteed to occur infinitely often for all traces.

So to check that a progress property holds, LTSA considers each terminal set T. It checks, for each action in the progress property set, whether the action is used between two states in T.

If, for some T, none of the actions in the progress property set occur as transitions in T, the property does not hold.

Progress analysis in LTSA

Now we can decode the error message generated by LTSA for the `TWOCOIN` process.

`Trace to terminal set of states`: gives a trace that leads to a terminal set void of the required actions.

`Cycle in terminal set`: gives some cycle of actions in that terminal set.

`Actions in terminal set`: gives the set of actions used in the terminal set.

If no progress property is specified, LTSA uses the default progress property, which is that every action in the alphabet of the system occurs infinitely often.

Example: readers/writers problem

The readers-writers problem is a problem in shared-access databases.

Database systems typically allow access from many processes at a time. Each thread is either a “reader”, which only reads from the database, or a “writer” that writes to the database.

Writers must have exclusive access to the database when accessing it (or more precisely, exclusive access to a record).

If there are no writers accessing the database, multiple readers should be able to access the database concurrently.

Example: readers/writers problem

In modelling concurrency, we are interested only in the actions related to the shared access database, and we abstract away the rest. The important actions for the reader and writer processes are the acquisitions and releases of a lock on the database:

```
set Actions = {acquireRead, releaseRead,
               acquireWrite, releaseWrite}
```

We have not encountered set declarations before. They are just like constant declarations, except they stand for a set of action names. We will use this set more than once in the model, so it is prudent to declare it as a set, rather than enumerate it each time we need to use this set of actions.

The reader and writer processes

Modelling the reader and writer processes is straightforward. The reader (writer) acquires the read (write) lock, examines (modifies) the data, and then releases the read (write) lock:

```
READER
 = (acquireRead -> examine -> releaseRead -> READER)
    +Actions.

WRITER
 = (acquireWrite -> modify -> releaseWrite -> WRITER)
    +Actions.
```

We have referenced the set `Actions` to add to both processes' alphabets. This ensures that prefixed actions of the `READER` cannot be interspersed freely, and similarly for the `WRITER` (see Lecture Con.08).

Locking the database

Access to the database is restricted via a read/write lock. A read lock can be acquired if no process is writing to the database, and a write lock can be acquired is no process is reading or writing. Multiple processes are permitted to read, but only one process can write.

```
const False = 0
const True  = 1
range Bool  = False..True
const Nread = 3             // Maximum readers
const Nwrite= 2             // Maximum writers
```

Locking the database

```
RW_LOCK = RW[0][False],
RW[readers:0..Nread][writing:Bool]
  = // When no other process is writing, we can read
    ( when (!writing)
        // Increment number of processes reading
        acquireRead -> RW[readers+1][writing]
    | // Upon release of a read lock,
      // decrement the number of readers
      releaseRead -> RW[readers-1][writing]
    | // If no processes are reading or writing,
      // a writer can acquire the write lock
      when (readers == 0 && !writing)
        acquireWrite -> RW[readers][True]
    | releaseWrite -> RW[readers][False]
    ).
```

Composing the readers, writers, and lock

Here is the composition of the readers, writers, and lock:

```
||READERS_WRITERS
  = (  reader[1..Nread]:READER
    || writer[1..Nwrite]:WRITER
    || {reader[1..Nread],writer[1..Nwrite]}::RW_LOCK
    ).
```

This composes `Nread` number of `READER` processes and `Nwrite` number of `WRITER` processes with a `RW_LOCK` process containing prefix labels for the readers and writers.

Composing the readers, writers, and lock

You may be concerned that the prefix labelling

```
{reader[1..Nread],writer[1..Nwrite]}::RW_LOCK
```

creates a process that allows one process to acquire a lock, but another process to release it.

However, the prefixed `READER` and `WRITER` processes do not allow such a transition, so the composite process `READERS_WRITERS` does not permit it either.

A readers/writers safety property

The safety property should express that a writer can only acquire the write lock when no other process is reading or writing, and a reader can only acquire a read lock when no process is writing:

```
property SAFE_RW
  = ( acquireRead -> READING[1]
    | acquireWrite -> WRITING
    ),
READING[i:1..Nread]
  = ( acquireRead -> READING[i+1]
    | when(i>1)  releaseRead -> READING[i-1]
    | when(i==1) releaseRead -> SAFE_RW
    ),
WRITING
  = (releaseWrite -> SAFE_RW).
```

A readers/writers safety property

This states that, when `acquireWrite` is observed, no other lock actions can occur until a `releaseWrite`. In addition, when an `acquireRead` is observed, only reads can be performed. The `READING` process keeps track of the number of processes reading.

We can now combine with the safety claim:

```
||SAFE_READERS_WRITERS
  = (   READERS_WRITERS
    ||  {reader[1..Nread],writer[1..Nwrite]}::SAFE_RW
    ).
```

Running this through the LTS analyser reveals no violations.

A readers/writers progress property

The desired progress property for the readers/writers example is obvious. All readers should eventually be able to read from the database, and all writers should eventually be able to write:

```
progress WRITE[i:1..Nwrite]
  = {writer[i].acquireWrite}
progress READ[i:1..Nwrite]
  = {reader[i].acquireRead}
```

Actually this says that some reader should eventually read, and some writer should eventually write; but since readers are identical (as are writers), that is probably fine.

Running this through the LTS analyser reveals no progress violations.

Progress in a stressed system

LTSA's analysis relies on the fair choice assumption. LTSA assumes that all options in all choices will eventually be taken.

However, in the readers/writers example, some choices are really only enabled if the database can keep up with requests.

Intuitively, if a reader gets access, and subsequently, there is always more than one reader reading the database, all writers will starve.

We can confirm this by simulating the stressed system, using so-called action priority.

FSP – Action priority

FSP has operators for action priority. These allow us to express a form of scheduling policy for a model.

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

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

This means that when there is a choice between an action in the set `a1,...,aN` and an action not in this set, the action in the set will be chosen, in case of high priority (for low priority the choice is opposite).

FSP – Action priority

This priority operator will actually remove some transitions from the process, so that the process `HIGH`:

```
P = (a -> b -> P | c -> d -> P).
||HIGH = P<<{a}.
```

is equivalent to:

```
HIGH = (a -> b -> HIGH).
```

Action priority for readers/writers

For the readers/writers system, we want to see what happens if the system is under pressure from many requests, as his would tend to give readers the advantage.

For this, we give the release actions a lower priority:

```
||RW_PROGRESS
  = READERS_WRITERS>>{reader[1..Nread].releaseRead,
                      writer[1..Nread].releaseWrite}.
```

Action priority for readers/writers

Running this through the LTSA analyser results in the following:

```
Progress violation: WRITE.1 WRITE.2
Trace to terminal set of states:
    reader.2.acquireRead
    reader.2.examine
Cycle in terminal set:
    reader.1.acquireRead
    reader.1.examine
    reader.1.releaseRead
Actions in terminal set:
    reader[1..2].{acquireRead, examine, releaseRead}
```

So if reader 2 acquires a read lock and examines, reader 1 can continually acquire a lock, examine, and release the lock before reader 2 releases its lock. This will starve all writers.

Revising readers/writers

To fix the problem, we need to make sure writers do not starve.

One step is to add an action `requestWrite` for writers, and a parameter `waitingW` to `RW_LOCK` that records how many writers are waiting. This is to ensure than readers can only acquire read locks if there are no writers waiting.

This will prevent writer starvation, but now readers may starve!

To prevent reader starvation, we add a boolean parameter `rTurn` to `RW_LOCK`, which allows readers to acquire locks if a writer has just had a turn.

Revising readers/writers

```
RW_LOCK
  = RW[0][False][0][False],
RW[readers:0..Nread][writing:Bool][waitingW:0..Nwrite][rTurn:Bool]
  = ( when (!writing && (waitingW == 0 || rTurn))
        acquireRead -> RW[readers+1][writing][waitingW][rTurn]
    | releaseRead -> RW[readers-1][writing][waitingW][False]
    | when (readers == 0 && !writing)
        acquireWrite -> RW[readers][True][waitingW-1][rTurn]
    | releaseWrite -> RW[readers][False][waitingW][True]
    | requestWrite -> RW[readers][writing][waitingW+1][rTurn]
    ).
```

Running this through the LTSA analyser now reveals no violations.