Checking Liveness

Vesal Vojdani
University of Tartu
Agenda for TLA+

• The exam will have simple processes that can and will be described in natural language.

• You will be asked to determine if something is a liveness or safety property and write down the temporal logic formula.

• You should be able to produce a simple counter-example trace (our states will have only one or two variables only).

• Next week, we will do the homework. The process will be a bit more complex, so we start solving it together. You only have to understand and describe the trace as homework.
Safety or Liveness?

• Variable “x” is always non-negative.

• Variable “x” will eventually be non-negative.

• It’s not the case that “x” is always non-negative!

• We will never acquire the same lock twice.

• Any locks that is acquired is also released.
Safety or Liveness?

- Variable “x” is always non-negative.
- Variable “x” will eventually be non-negative.
- It’s not the case that “x” is always non-negative!
- We will never acquire the same lock twice.
- Any locks that is acquired is also released.
Negation

• Variable “x” is always non-negative.

• If we just negate the invariant:  
  Variable “x” is always negative.

• It’s not the case that “x” is always non-negative =  
  The variable “x” is eventually negative.

• It helps if we are a bit more precise…
We only consider the fragment used in TLA.

You have formulas in (predicate logic):

- $x = 0$

Eventually (sometimes called finally):

- $\Diamond (x = 0)$

Always (sometimes called globally):

- $\Box (x = 0)$
A trace is a (possibly infinite) sequence of states. You should always imagine that you are here, looking into the future!
Example: $x = 0$

Assume a process that changes the variable “$x$”. Does the above trace satisfy the formula $x = 0$?

Yes, because it holds at the beginning of the trace!
Eventually

A trace satisfies $\Diamond \phi$ if there exists a future time (a suffix trace) that satisfies the formula $\phi$
Always

A trace satisfies □φ if all of its suffix traces satisfy the formula φ.
Example: \( \diamondsuit(x = 1) \)

Yes, because it holds at the beginning of this (suffix) trace!
$\square (\Diamond (x = 1))$

"x" will always be eventually 1?
Eventually $x = 1$ holds here!

Because trace starting here satisfies $x = 1$

$\square (\Diamond (x = 1))$

“$x$” will always be eventually 1?
Eventually $x = 1$ holds here!

0 $\rightarrow$ 1 $\rightarrow$ 0 $\rightarrow$ 1 $\rightarrow$ 0 $\rightarrow$ ... 

$\square (\Diamond (x = 1))$

“x” will always be eventually 1?
Eventually $x = 1$ holds here!

$$0 \rightarrow 1 \rightarrow 0 \rightarrow 1 \rightarrow 0 \rightarrow \ldots$$

$\square(\Diamond(x = 1))$

“$x$” will always be eventually 1?
Eventually $x = 1$ holds here!

$0 \rightarrow 1 \rightarrow 0 \rightarrow 1 \rightarrow 0 \rightarrow \ldots$

$\square(\Diamond(x = 1))$

“$x$” will always be eventually 1?
"x" will always be eventually 1?

Eventually $x = 1$ holds here!
$0 \rightarrow 1 \rightarrow 0 \rightarrow 1 \rightarrow 0 \rightarrow \ldots$

$\diamond (\square (x = 1))$

"$x$" will eventually be always 1?
No, this requires that we have a suffix that satisfies the formula “x is always 1”.

\[ \Diamond (\Box (x = 1)) \]
This one does satisfy it!
Safety and Liveness

- **Safety**: something bad never happens.

- **Liveness**: something good will happen.

- The key difference: if a (finite) prefix of a trace violates the property, can the entire trace still satisfy it?

- Or ask yourself: does shutting down the system prevent the property from being satisfied?
Safety and Liveness

- **Safety**: something bad never happens.
- **Liveness**: something good will happen.

The key difference: if a (finite) prefix of a trace violates a property, can the entire trace still satisfy it?

Or ask yourself: does shutting down the system prevent the property from being satisfied?

Security is a safety property, and indeed, No Internet is the best security! (Pages actually loading is a liveness property.)

In either case, yes = liveness
A process satisfies a formula if all its possible traces satisfy that formula!
NextColor(c) == CASE c = "red" -> "green"
    □ c = "green" -> "red"

(*--algorithm traffic
variables
    at_light = TRUE,
    light = "red";

process light = "light"
begin Cycle:
    while at_light do
        light := NextColor(light);
    end while;
end process;

end algorithm;*)

Example in TLA
Wayne, chapter 6 (traffic.tla)
Do It!

- Create a model and add/check "Termination" under Model Overview > What to Check? > Properties.

- Make both processes fair. (Run Again! Note the difference!)

- Make the car process strongly fair.
Fairness

- Stuttering: we can always fail to take a transition! (This can model hardware failure)

- A weakly fair action will, if it stays enabled, eventually happen.

- A strongly fair action, if it’s repeatedly enabled, will eventually happen.
Processes can still stutter!

Strong fairness only ensures that if a process can make some transition infinitely often, it will make some transition.

(* --algorithm stutter

variables state = 0;

fair+ process donothing = "nothing"
begin Loop:
  state := 0;
  goto Loop
end process

end algorithm*)

Processes can still stutter!

Strong fairness only ensures that if a process can make some transition infinitely often, it will make some transition.
fair+ process dosomething = "something"
begin Loop:
  either state := 0 or state := 1 end either;
goto Loop
end process

Fairness forces a transition

This process will satisfy ◇(state = 1).
The state has to change, so we will do state := 1.
Fairness forces a transition

This process will not satisfy $\diamond (\text{state} = 1)$
The trace $0 \rightarrow 2 \rightarrow 0 \rightarrow 2 \rightarrow 0 \rightarrow \ldots$ is still possible.

fair+ process dosomething = "something"
begin Loop:
either state := 0 or state := 1 or state := 2 end either;
goto Loop
end process

Not necessarily the one I want!
For the exam

• We have a looping process with a single variable. At each step, it sets the variable non-deterministically to one of \{1,2,3\}. You may assume the process is fair.

• State that the value of “x” is always positive.

• State that the value of “x” will eventually be one.

• Which of the above is safety property and which is a liveness property?

• Which of the above properties are satisfied by the describe process. If note satisfied, provide a counter-example trace.
Question Time

- What about fairness and safety conditions?
- Why didn’t we need it for invariant checking?
- This is important. Do not proceed unless you can figure this one out on your own.
- (The answer is on previous slides...)
Mutual Exclusion
EXTENDS TLC, Integers
CONSTANT Threads
(*--algorithm dekker
variables flag = [t \in Threads \rightarrow FALSE]

process thread \in Threads
begin
  P1: flag[self] := TRUE;
  P2: await \A t \in Threads \setminus \{self\}: flag[t] = FALSE;
  CS: skip;
  P3: flag[self] := FALSE;
end process;
end algorithm; *)

Mutual Exclusion
(First attempt)
Safety condition

- Only one thread should enter the critical section at a time.
- Write this invariant yourself and run the model checker with and without the await-condition.
- It should fail without the check.
- And will deadlock with the check!
process thread \in Threads
begin
  P1: flag[self] := TRUE;
  P2: \textbf{await} \ \ \ \forall t \in Threads \ \{self\}: flag[t] = FALSE;
    CS: skip;
  P3: flag[self] := FALSE;
end process;

\textbf{Simple Fix!}

Busy wait and signal our interest briefly. Does this work?
TLC says yes!

- Okay, but today’s topic is liveness.

- Let’s ask another question: will both threads eventually make it to the critical section?

- Write this down as a TLA formula and run this “property”!

- Make sure your process is “fair”!

- Remember: [ ] means always and <> eventually.
Livelock!
(note the “back to state 3”)
variables
flag = [t \in Threads |-> FALSE],
next_thread \in Threads;

fair process thread \in Threads
begin
P1: flag[self] := TRUE;
P2: while \E t \in Threads \ {self}: flag[t] do
P2_1:
  if next_thread /= self then
    P2_1_1: flag[self] := FALSE;
    P2_1_2: await next_thread = self;
    P2_1_3: flag[self] := TRUE;
  end if;
end while;
CS: skip;
P3: with t \in Threads \ {self} do
  next_thread := t;
end with;
P4: flag[self] := FALSE;
end process;

Dekker’s algorithm
We saw examples of...

- Safety property (mutual exclusion)
  \[ \forall t1, t2 \in Threads : \]
  \[ pc[t1] = "CS" \land pc[t2] = "CS" \Rightarrow t1 = t2 \]

- Liveness property
  \[ \forall t \in Threads : \Diamond (pc[t] = "CS") \]

- Deadlock and Livelock