TFE4171 Exercise 4 Report

Morten Paulsen
Magnus Ramsfjell

March 28, 2020

1 Lab 1 - Formal verification of readserial

1.1 State Transition Graph

The **readserial** controller has two states: **IDLE** and **READDATA**. It transitions according to the graph in Figure 1.

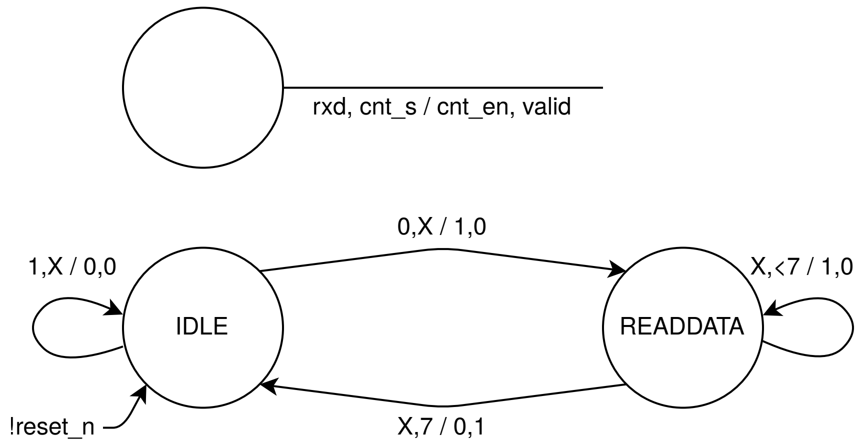

Figure 1: State transition graph for the **readserial** controller

1.2 reset_sequence

```
sequence reset_sequence;
        !reset_n;
endsequence;

property reset;
        reset_sequence |=>

        t ##0 state_s == IDLE and
        t ##0 !valid;
endproperty;

a_reset: assert property (@(posedge clk) reset);
```

Listing 1: Implementation of the **reset** property

1.3 stay_in_idle

```
sequence idle_sequence;
        rxd and state_s == IDLE;
endsequence;

property stay_in_idle;
        t ##0 idle_sequence

        implies

        t ##1 state_s == IDLE;
endproperty;

a_stay_in_idle: assert property (@(posedge clk) stay_in_idle);
```

Listing 2: Implementation of the **stay_in_idle** property

1.4 read_byte

```
sequence start_sequence;
        !rxd and state_s == IDLE;
endsequence;

property read_byte;
        logic [7:0] tmp;

        t ##0 start_sequence and
        t ##8 set_freeze(tmp,
                {
                        $past(rxd, 7),
                        $past(rxd, 6),
                        $past(rxd, 5),
                        $past(rxd, 4),
                        $past(rxd, 3),
                        $past(rxd, 2),
                        $past(rxd, 1),
                         rxd
                })

        implies

        t ##1 (state_s == READDATA and !valid) and
        t ##2 (state_s == READDATA and !valid) and
        t ##3 (state_s == READDATA and !valid) and
        t ##4 (state_s == READDATA and !valid) and
        t ##5 (state_s == READDATA and !valid) and
        t ##6 (state_s == READDATA and !valid) and
        t ##7 (state_s == READDATA and !valid) and
        t ##8 (state_s == READDATA and !valid) and
        t ##9 data == tmp and
        t ##9 valid and
        t ##9 state_s == IDLE;
endproperty;

a_read_byte: assert property (@(posedge clk) disable iff (!reset_n) read_byte);
```

Listing 3: Implementation of the **read_byte** property

1.5 read_byte, basic IPC solver

```
sequence start_sequence;
        !rxd and state_s == IDLE;
endsequence;

sequence in_idle_counter_is_0;
        state_s == IDLE and cnt_s == 0;
endsequence;

sequence in_idle_counter_not_enabled;
        state_s == IDLE and !cnt_en;
endsequence;

property read_byte;
        logic [7:0] tmp;

        t ##0 start_sequence and
        t ##0 in_idle_counter_is_0 and
        t ##0 in_idle_counter_not_enabled and
        t ##8 set_freeze(tmp,
                {
                        $past(rxd, 7),
                        $past(rxd, 6),
                        $past(rxd, 5),
                        $past(rxd, 4),
                        $past(rxd, 3),
                        $past(rxd, 2),
                        $past(rxd, 1),
                         rxd
                })

        implies

        t ##1 (state_s == READDATA and !valid) and
        t ##2 (state_s == READDATA and !valid) and
        t ##3 (state_s == READDATA and !valid) and
        t ##4 (state_s == READDATA and !valid) and
        t ##5 (state_s == READDATA and !valid) and
        t ##6 (state_s == READDATA and !valid) and
        t ##7 (state_s == READDATA and !valid) and
        t ##8 (state_s == READDATA and !valid) and
        t ##9 data == tmp and
        t ##9 valid and
        t ##9 state_s == IDLE;
endproperty;

a_read_byte: assert property (@(posedge clk) disable iff (!reset_n) read_byte);
```

Listing 4: Implementation of the updated **read_byte** property

The sequence `in_idle_counter_is_0` is added to prevent OneSpin from assuming that `cnt_s` can be any value, and not only 0. The counterexample is shown in Fig. 2

The sequence `in_idle_counter_not_enabled` is added to prevent OneSpin from assuming that `cnt_en` can be enabled when `state_s` is `IDLE`. The counterexample is shown in Fig. 3.

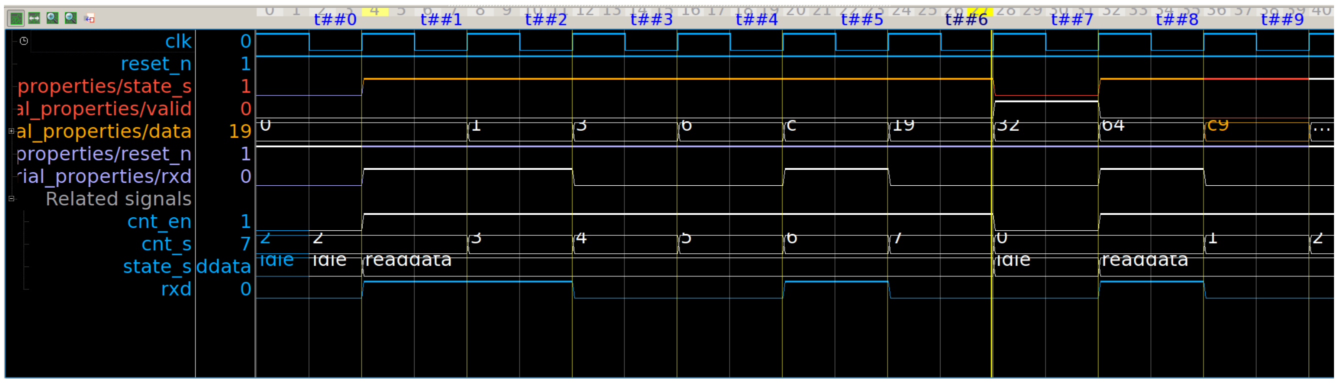

Figure 2: Counterexample before adding in_idle_counter_is_0.

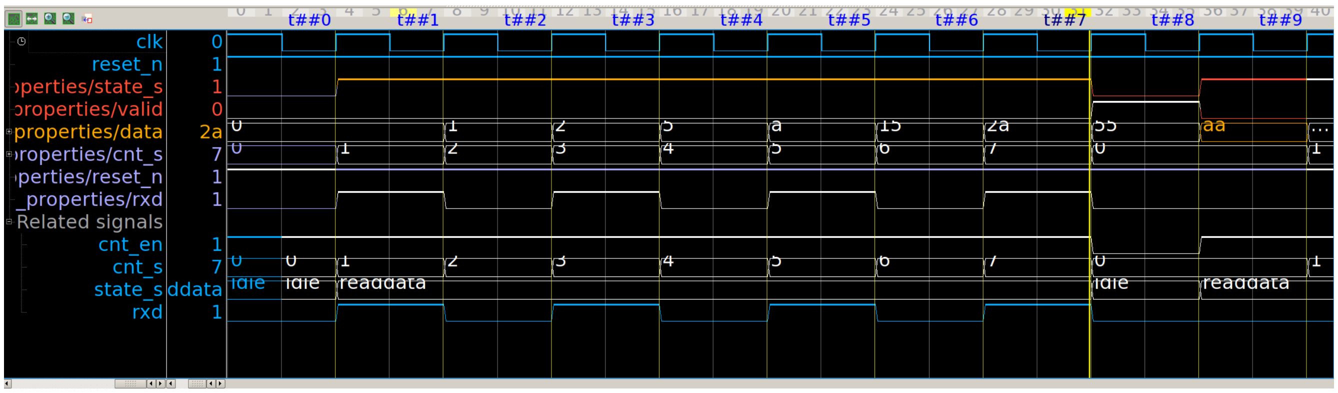

Figure 3: Counterexample before adding in_idle_counter_not_enabled.

1.6 read_byte, inductive proof for in_idle_counter_not_enabled

1.6.1 in_idle_counter_not_enabled__step

```
property in_idle_counter_not_enabled__step;
        t ##0 (in_idle_counter_not_enabled and rxd)

        implies

        t ##1 in_idle_counter_not_enabled;
endproperty;

a_in_idle_counter_not_enabled__step: assert property (@(posedge clk) disable iff (!reset_n) in_idle_counter_not_enabled__step);
```

Listing 5: Step proof for in_idle_counter_not_enabled.

1.6.2 in_idle_counter_not_enabled__base

```
property in_idle_counter_not_enabled__base;
        t ##0 reset_sequence

        implies

        t ##1 in_idle_counter_not_enabled;
endproperty;

a_in_idle_counter_not_enabled__base: assert property (@(posedge clk) in_idle_counter_not_enabled__base);
```

Listing 6: Base proof for in_idle_counter_not_enabled.

1.7 read_byte, inductive proof for in_idle_counter_is_0

1.7.1 in_idle_counter_is_0, first attempt

```
property in_idle_counter_is_0__step;
        t ##0 (in_idle_counter_is_0 and rxd)

        implies

        t ##1 in_idle_counter_is_0;
endproperty;

a_in_idle_counter_is_0__step: assert property (@(posedge clk) disable iff (!reset_n) in_idle_counter_is_0__step);
```

Listing 7: Step proof for in_idle_counter_is_0.

```
property in_idle_counter_is_0__base;
        t ##0 reset_sequence

        implies

        t ##1 in_idle_counter_is_0;
endproperty;

a_in_idle_counter_is_0__base: assert property (@(posedge clk) in_idle_counter_is_0__base);
```

Listing 8: Base proof for in_idle_counter_is_0.

The counterexample given by OneSpin when we used the single-step inductive proof is shown in Fig. 4.

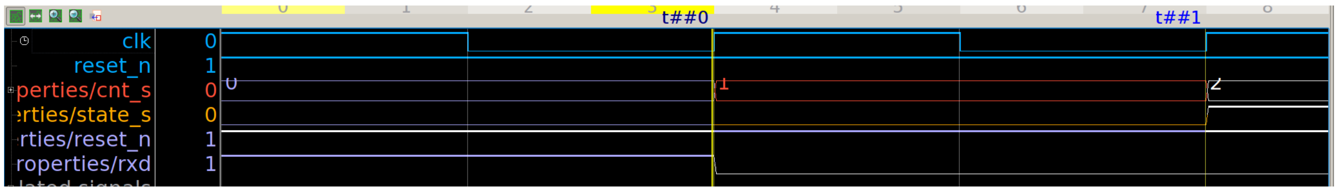

Figure 4: Counterexample for the inductive proof for in_idle_counter_is_0__step.

The counterexample shows an example where `cnt_s` is incremented when `state_s == IDLE`. This happens since the assumption is only based on one clock cycle and there is not made any assumption that `cnt_s` is to be stable between cycles. This can be prevented by introducing an extra step in the assumption, if we assume that the sequence holds in two cycles (`##0` and `##1`) it must also hold in cycle `##2`, as the only way to find a counterexample to the implication is to break the assumption.

For this counterexample specifically, the `in_idle_counter_is_0` sequence holds for the first cycle (`t ##0`), but fails at the next cycle because `cnt_s` has increased. This counterexample is found by having the internal signal `cnt_en` set high—which has not been proved to be unreachable in this property—which started the counter. The counter then increments to 1 at `t ##1`, causing the property to fail. Solving this requires either appending `and !cnt_en` to the `in_idle_counter_is_0` sequence (which somewhat defeats the purpose of the `in_idle_counter_not_enabled`), adding the
`in_idle_counter_not_enabled` sequence to the `in_idle_counter_is_0__step` property assumption, or by assuming that the `in_idle_counter_is_0` sequence holds for two consecutive cycles. The former could be regarded as the "proper" solution, as it implicitly captures the fact that the count signal is not enabled (which it never should have been in this case) by using two cycles where `cnt_s` did not change.

1.7.2 in_idle_counter_is_0, second attempt

```
property in_idle_counter_is_0__step;
        t ##0 (in_idle_counter_is_0 and rxd) and
        t ##1 (in_idle_counter_is_0 and rxd)

        implies

        t ##2 in_idle_counter_is_0;
endproperty;

a_in_idle_counter_is_0__step: assert property (@(posedge clk) disable iff (!reset_n) in_idle_counter_is_0__step);
```

Listing 9: Step proof for in_idle_counter_is_0.

```
property in_idle_counter_is_0__base;
        t ##0 reset_sequence and
        t ##1 rxd

        implies

        t ##1 in_idle_counter_is_0 and
        t ##2 in_idle_counter_is_0;
endproperty;

a_in_idle_counter_is_0__base: assert property (@(posedge clk) in_idle_counter_is_0__base);
```

Listing 10: Base proof for in_idle_counter_is_0.

To prove the properties we need to have `n=2` as this ensures that `cnt_s` is kept at 0. `cnt_s` has no change between `t ##0` and `t ##1` nor `t ##1` and `t ##2`. This was not taken consideration in the first attempt.

2 Lab 2 - Completeness checking of readserial

```
// @lang=vli @ts=2

completeness readserial;
disable iff: (!reset_n);
inputs: reset_n, rxd;

determination_requirements:
  determined(valid);
  if (valid) determined(data); endif;

reset_property:
  sva/inst_readserial_properties/ops/a_reset;
property_graph:
  sva/inst_readserial_properties/ops/a_reset -> sva/inst_readserial_properties/ops/a_read_byte;
  sva/inst_readserial_properties/ops/a_reset -> sva/inst_readserial_properties/ops/a_stay_in_idle;
  sva/inst_readserial_properties/ops/a_stay_in_idle -> sva/inst_readserial_properties/ops/a_read_byte;
  sva/inst_readserial_properties/ops/a_stay_in_idle -> sva/inst_readserial_properties/ops/a_stay_in_idle;
  sva/inst_readserial_properties/ops/a_read_byte -> sva/inst_readserial_properties/ops/a_read_byte;
  sva/inst_readserial_properties/ops/a_read_byte -> sva/inst_readserial_properties/ops/a_stay_in_idle;
end completeness;
```

Listing 11: The modified completeness.gfv

In Listing 11 we added the following pairs of operation property and direct successor operation property:

```
sva/inst_readserial_properties/ops/a_reset -> sva/inst_readserial_properties/ops/a_stay_in_idle;
sva/inst_readserial_properties/ops/a_stay_in_idle -> sva/inst_readserial_properties/ops/a_read_byte;
sva/inst_readserial_properties/ops/a_stay_in_idle -> sva/inst_readserial_properties/ops/a_stay_in_idle;
sva/inst_readserial_properties/ops/a_read_byte -> sva/inst_readserial_properties/ops/a_stay_in_idle;
```

These pairs makes the set of possible property transitions complete.

We also had to make some modifications to the properties themselves for the end point of a operation property to align with the start point of the direct successor operation property. The modifications was to make sure that the signals `valid`, `state_s`, `cnt_en` and `cnt_s` was explicitly set both at the beginning and at the end of the properties `a_stay_in_idle` and `read_byte`. The modifications are not shown here, but can be found in the appended file `readserial.tda`.