Lab 2 – NuSMV and liveness properties

1 Preliminary Exercise

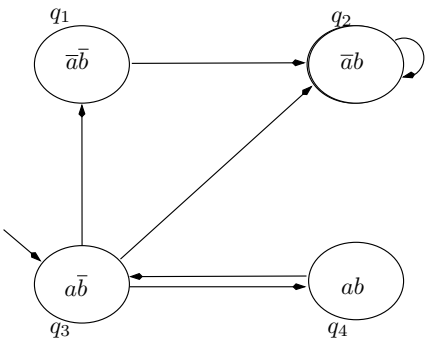

Figure 1: Model for Exercise 1.

Create a NuSMV model for the system shown in Fig 1. For each of the LTL formulas φ below, use NuSMV to *(i)* determine whether the system satisfies the formula φ, and *(ii)* persuade NuSMV to exhibit some execution which satisfies φ. Check that the answers you get with NuSMV correspond to your own understanding of the model and the formulas. In particular, you should understand why both φ and $\neg\varphi$ can be false.

(a) $\Diamond b$
(b) $\Box a$
(c) $a \,\mathcal{U}\, b$
(d) $a \,\mathcal{U}\, \bigcirc b$
(e) $\Box\Diamond b$
(f) $\Diamond\Box b$

Hints:

- The simplest solution is to create a NuSMV model of the state machine that uses 1 state variable with 4 values, one for each of the states of the state machine. Then use `DEFINE` assignments to specify in which states the base formulas a and b are true. An alternative approach that can yield a smaller model, but that can be slightly less straightforward, is to introduce 2 state variables, one for a, one for b.
- For (ii), consider what NuSMV does if you try to verify $\neg\varphi$
- The NuSMV syntax for `LTLSPEC` is the following:

$\wedge$ (and):	`&`	$\vee$ (or):	`|`	$\neg$ (not):	`!`	$\rightarrow$ (implies):	`->`
$\Box$ (always):	`G`	$\Diamond$ (eventually):	`F`	$\bigcirc$ (next):	`X`	$\mathcal{U}$ (until):	`U`

  When writing NuSMV formulas, note that the precedence of LTL operators (stronger to weaker) is `F G X ! U & | ->`.

2 NuSMV: verifying a FIFO

For this exercise, you verify properties of a model of a FIFO digital circuit. A block diagram of the FIFO (First In First Out) circuit is shown in Figure 2.

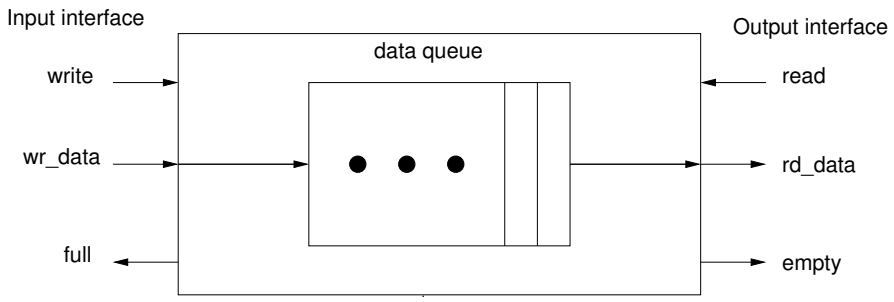

Figure 2: Block diagram of the FIFO.

A FIFO is a variable-length queue of data words. It has two interfaces, one input interface for adding words to one end of the queue and one output interface for reading and removing words from the other end of the queue.

To add or write an element to the FIFO, the data is presented on the `wr_data` input and the Boolean signal `write` is asserted (set to true). Provided that the FIFO is not full, the element is added to the queue on the transition to the next step. The FIFO has a maximum number of elements it can hold in the queue at any one time. The Boolean output `full` of the FIFO indicates whether or not it currently holds the maximum number.

The output `rd_data` of the FIFO shows the current last element in the FIFO's internal queue, provided that the queue is not empty. The queue being empty is signalled by the Boolean output `empty` being set to true. If the Boolean signal `read` is set to true and the queue is not empty, on the transition to the next step the current last element in the queue is removed and the element behind it (if any) then appears on the FIFO output `rd_data`.

The file `fifo.smv` contains the FIFO model. For simplicity and to ensure rapid NuSMV execution times, we set the `DEPTH` constant for the maximum number of elements in the queue to 5 and the `WIDTH` constant for the element size to 1 bit. A real system would often use larger values for both parameters.

Internally, the FIFO uses a circular buffer to implement the queue. This consists of an array `buffer` of size `DEPTH` and two pointers into this array, the read pointer `rd_p` and the write pointer `wr_p`. If the queue is not empty, the read pointer points to the last element of the queue. If the queue is not full, the write pointer points to the position to write the next input element. When a new element is written into the queue, the write pointer is incremented, wrapping it around as necessary. When an element in the queue is removed, the read points is incremented, wrapping it around as necessary. See Figure 3 for two examples of the internal configuration of the FIFO when the queue holds the words `w0`, `w1` and `w2`, added in that order.

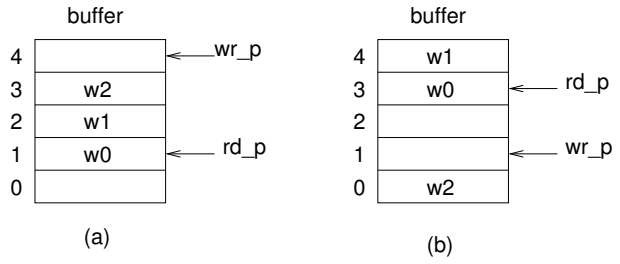

Figure 3: Example of internal configurations of the FIFO.

In the provided implementation of the FIFO, the queue could be either empty or full when the two pointers are equal. The design uses the state variable `empty` to distinguish between these two cases.

2.1 Properties to verify

In the provided template file `fifo-properties.smv`, add formulas for the LTL properties requested below. Verify your properties with NuSMV by running the command

```
NuSMV -pre cpp fifo.smv
```

The `fifo.smv` file brings in the `fifo-properties.smv` file using a preprocessor directive `#include`. The `-pre cpp` option of NuSMV is necessary to ensure it runs the C preprocessor on `fifo.smv` in order to interpret this directive. You will get warnings until you fill in the LTL specifications.

Write LTL formulas for the following properties (all true in `fifo.smv`):

(a) *It is never the case that the FIFO indicates simultaneously it is both empty and full*

(b) *If* `write` *is persistend and* `read` *is not recurrent, then the FIFO eventually becomes full.*

(c) *At any time, if a* 1 *is presented to the FIFO data input and* `write` *is asserted, then eventually a* 1 *will appear on the FIFO data output*

with further reasonable assumptions added after the *if* concerning FIFO signals such as `read`, `empty` or `full`, to ensure the property checks true. Consult the NuSMV user guide for information on *word constants* to be able to refer to the number 1.

(d) the same property as in (c), except that it is phrased to hold for any data value, not just the value 1. Use the 'frozen variable' `data1` to do this. Consult the NuSMV user guide for documentation on *frozen variables* (also sometimes known in temporal logic as rigid variables).

(e) the same property as in (d), except that, in addition, it requires the `empty` output of the FIFO to be set to false at all times *inbetween* the time the write of the data is set up and the time the data can first be read out, but not actually at either of these times. You may take advantage of the fact that the earliest we expect the data to appear is the step after it is written.

(f) a similar property to that for (d), except that it assumes that two possibly-distinct data values are input on consecutive steps, and checks for the same two values appearing on the output on consecutive steps. Use the provided frozen variables `data1` and `data2` to refer to the two data values.

2.2 Bug correction

The FIFO has a bug. In this part you discover and fix it.

1. In the indicated place in `fifo-properties.smv`, write an LTL property that checks that

   always, if the FIFO indicates it is empty, then the read and write pointers are equal.

   NuSMV should find it false and show a counter-example.

2. Make a copy of `fifo.smv` called `fifo-fixed.smv`. Make changes to the code in the main module in the `fifo-fixed.smv` file to fix this bug. Your changes should address the general problem identified by this bug.

3 Principles of LTL model checking

As remarked in lecture, in LTL model checking of a formula φ, one constructs a Büchi monitor for $\neg\varphi$ which accepts just those traces ρ that satisfy $\neg\varphi$. The formula is then true just when there are no accepting executions in the composition of the system with the monitor.

Let φ be the LTL property $\Box\Diamond$`write` $\rightarrow \Diamond$`full`.

(a) Write $\neg\varphi$ in a normalised form, where the negations are pushed inwards so they just surround atomic formulas and the only binary logical connectives used are $\wedge$ and $\vee$. This should simplify the writing of a Büchi automaton for $\neg\varphi$.

(b) Write a NuSMV module that emulates a Büchi monitor for $\neg\varphi$. Hint: you should not need an automaton with more than 3 or 4 states.

(c) Write an LTL property that captures the acceptance condition of the Büchi automaton, that, if true, indicates that there are no accepting runs of the monitor.

Insert your solution into the file `fifo-ltlmc.smv` in the indicated positions at the start. This file include a copy of the module from `fifo.smv`, but with the `main` module renamed to `system` and a new `main` module that composes the system with the negated formula automaton.