Course "Formal Methods"
Lab Test

Roberto Sebastiani
DISI, Università di Trento, Italy

September 06^{th}, 2018

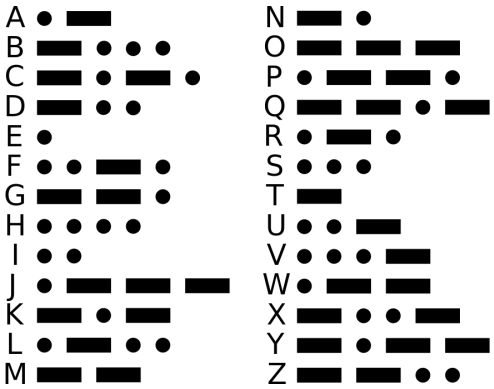
769857918

[COPY WITH SOLUTIONS]

1 Spin

Model a system comprised of the following processes:

- process `encoder`: listens for incoming letters, encoded as `bytes`, on an input channel. For each incoming letter which it recognizes, it forwards its corresponding *Morse* code through its output channel. Use the following table as reference:



  Unrecognized characters are ignored. After each incoming character, recognized or not, the `encoder` forwards a special `NEWLINE` message through its output channel.

- process `printer`: listens for an incoming `DOT`, `LINE` or `NEWLINE` message and it prints the corresponding symbol on the console. A `DOT` message is printed as a `'.'`, a `LINE` message is printed as a `'-'` and a `NEWLINE` message is printed as a newline, so that any further output text is printed on a different line.

- process `init`: initializes an instance of `encoder` and an instance of `printer`, so that the output of the `encoder` becomes the input of the `printer` instance. Then, it sends through the input channel of the `encoder` all the letters which make up the *name* and *surname* of the student taking this exam.

Simulate the system and visually verify that it prints your *name* and *surname* on the standard output, using a single line for each *Morse encoded* character. Ensure that all the processes correctly terminate when the last character of your surname has been printed.

Notes: a sequence of four dots, which corresponds to letter `H`, can be forwarded to the `printer` as four distinct `DOT` messages or a single `DOT` with a multiplicity factor equal to 4. You are free to choose the approach you want.

Solution:

```
mtype = { DOT, LINE, NEWLINE };

proctype encoder (chan chin, chou)
{
    byte cc;
    do
        :: chin?cc ->
            if
                :: cc == 'A' || cc == 'a' ->
                    chou!DOT(1); chou!LINE(1);
                :: cc == 'B' || cc == 'b' ->
                    chou!LINE(1); chou!DOT(3);
                :: cc == 'C' || cc == 'c' ->
                    chou!LINE(1); chou!DOT(1); chou!LINE(1); chou!DOT(1);
                :: cc == 'D' || cc == 'd' ->
                    chou!LINE(1); chou!DOT(2);
                :: cc == 'E' || cc == 'e' ->
                    chou!DOT(1);
                :: cc == 'F' || cc == 'f' ->
                    chou!DOT(2); chou!LINE(1); chou!DOT(1);
                :: cc == 'G' || cc == 'g' ->
                    chou!LINE(2); chou!DOT(1);
                :: cc == 'H' || cc == 'h' ->
                    chou!DOT(4);
                :: cc == 'I' || cc == 'i' ->
                    chou!DOT(2);
                :: cc == 'J' || cc == 'j' ->
                    chou!DOT(1); chou!LINE(3);
                :: cc == 'K' || cc == 'k' ->
                    chou!LINE(1); chou!DOT(1); chou!LINE(1);
                :: cc == 'L' || cc == 'l' ->
                    chou!DOT(1); chou!LINE(1); chou!DOT(2);
                :: cc == 'M' || cc == 'm' ->
                    chou!LINE(2);
                :: cc == 'N' || cc == 'n' ->
                    chou!LINE(1); chou!DOT(1);
                :: cc == 'O' || cc == 'o' ->
                    chou!LINE(3);
                :: cc == 'P' || cc == 'p' ->
                    chou!DOT(1); chou!LINE(2); chou!DOT(1);
                :: cc == 'Q' || cc == 'q' ->
                    chou!LINE(2); chou!DOT(1); chou!LINE(1);
                :: cc == 'R' || cc == 'r' ->
                    chou!DOT(1); chou!LINE(1); chou!DOT(1);
                :: cc == 'S' || cc == 's' ->
                    chou!DOT(3);
                :: cc == 'T' || cc == 't' ->
                    chou!LINE(1);
```

```
                :: cc == 'U' || cc == 'u' ->
                    chou!DOT(2); chou!LINE(1);
                :: cc == 'V' || cc == 'v' ->
                    chou!DOT(3); chou!LINE(1);
                :: cc == 'W' || cc == 'w' ->
                    chou!DOT(1); chou!LINE(2);
                :: cc == 'X' || cc == 'x' ->
                    chou!LINE(1); chou!DOT(2); chou!LINE(1);
                :: cc == 'Y' || cc == 'y' ->
                    chou!LINE(1); chou!DOT(1); chou!LINE(2);
                :: cc == 'Z' || cc == 'z' ->
                    chou!LINE(2); chou!DOT(2);
                :: else -> skip;
            fi;
            chou!NEWLINE(1);
        :: timeout -> break;
    od
}

proctype printer(chan chin)
{
    byte code, cc, idx;
    do
        :: chin?code(cc) ->
            for (idx: 0 .. cc - 1) {
                if
                    :: DOT == code ->
                        printf(".");
                    :: LINE == code ->
                        printf("-");
                    :: NEWLINE == code ->
                        printf("\n");
                    :: else -> skip;
                fi
            }
        :: timeout -> break;
    od;
}

init {
    chan main_to_encoder    = [1] of { byte };
    chan encoder_to_printer = [1] of { byte, byte };

    run encoder(main_to_encoder, encoder_to_printer);
    run printer(encoder_to_printer);

    main_to_encoder!'P'; main_to_encoder!'A'; main_to_encoder!'T'; main_to_encoder!'R';
    main_to_encoder!'I'; main_to_encoder!'C'; main_to_encoder!'K'; main_to_encoder!' ';
    main_to_encoder!'T'; main_to_encoder!'R'; main_to_encoder!'E'; main_to_encoder!'N';
    main_to_encoder!'T'; main_to_encoder!'I'; main_to_encoder!'N'; main_to_encoder!'\n';
}
```

2 nuXmv

Encode the following *sorting_network* procedure for an array of length 5 in NuSMVor nuXmv as a **module** `sorting_network(arr, len)`:

```
    def sorting_network(arr, len):
        i = 0
        j = 0
L00:    while (true):
L01:        if (!arr[i]):
L02:            j = len - 1
L03:            while (j > i):
L04:                if (arr[j]):
L05:                    swap(arr[i], arr[j])
L06:                    break
L07:                j = j - 1
L08:        if ((len - 1) == i):
L09:            break
L10:        i = i + 1
L11:    return                      # self-loop here!
```

Declare, inside the **main** module, the following variables:

- `arr`, an array of 5 Boolean elements, initialized to { `TRUE, FALSE, TRUE, FALSE, TRUE` }
- `sn` is an instance of `sorting_network` taking as input `arr` and 5

Declare, inside the **sorting_network** module, the following variables:

- `i`, with domain $0..len - 1$
- `j`, with domain $0..len - 1$
- `pc`, with domain { `L00, L01, L02, L03, L04, L05, L06, L07, L08, L09, L10, L11` }

Verify that all of the following properties, with the exception of the last one, are `true`:

- Sooner or later the execution reaches the final state `L11`
- In the final state, the content of the array `arr` is equal to { `T, T, T, F, F` }
- Sooner or later, the value of `i` and `j` will remain equal to 4 forever
- Invariant: `j` is never smaller than `i`

Hints. Ensure that the content of `arr` is not changed at any other line than `L05`, and that no other memory location than those indexed by `i` and `j` is changed while *swapping* the content of `arr[i]` with `arr[j]`. You **can exploit** the fact that the value of `i` and `j` does not change in the following state to simplify the encoding of the transition relation.

Solution:

```
MODULE main()
VAR
    arr : array 0..4 of boolean;
    sn  : mode(arr, 5);

INIT arr[0] = TRUE & arr[1] = FALSE & arr[2] = TRUE & arr[3] = FALSE & arr[4] = TRUE;

MODULE mode(arr, len)
VAR
    i   : 0..len - 1;
    j   : 0..len - 1;
    pc  : { L00, L01, L02, L03, L04, L05, L06, L07, L08, L09, L10, L11 };

INIT
    pc = L00 & i = 0 & j = 0;

ASSIGN
    next(pc) := case
        L00 = pc                 : L01;
        L01 = pc & !arr[i]       : L02;
        L01 = pc                 : L08;
        L02 = pc                 : L03;
        L03 = pc & j > i         : L04;
        L03 = pc                 : L08;
        L04 = pc & arr[j]        : L05;
        L04 = pc                 : L07;
        L05 = pc                 : L06;
        L06 = pc                 : L08;
        L07 = pc                 : L03;
        L08 = pc & (len - 1) = i : L09;
        L08 = pc                 : L10;
        L09 = pc                 : L11;
        L10 = pc                 : L00;
        L11 = pc                 : L11;
    esac;

    next(i) := case
        L10 = pc & i < len - 1 : i + 1;
        TRUE                   : i;
    esac;

    next(j) := case
        L02 = pc           : len - 1;
        L07 = pc & j > 0   : j - 1;
        TRUE               : j;
    esac;

TRANS (pc != L05 -> ((next(arr[0]) = arr[0]) & (next(arr[1]) = arr[1]) &
                     (next(arr[2]) = arr[2]) & (next(arr[3]) = arr[3]) &
                     (next(arr[4]) = arr[4])));
```

```
TRANS (pc = L05) -> ((next(arr[i]) = arr[j]) & (next(arr[j]) = arr[i]));
TRANS (pc = L05 & i != 0 & j != 0) -> (next(arr[0]) = arr[0]);
TRANS (pc = L05 & i != 1 & j != 1) -> (next(arr[1]) = arr[1]);
TRANS (pc = L05 & i != 2 & j != 2) -> (next(arr[2]) = arr[2]);
TRANS (pc = L05 & i != 3 & j != 3) -> (next(arr[3]) = arr[3]);
TRANS (pc = L05 & i != 4 & j != 4) -> (next(arr[4]) = arr[4]);

-- Sooner or later the execution reaches the final state
LTLSPEC F (pc = L11);

-- In the final state, the content of arr is { T, T, T, F, F }
LTLSPEC G (pc = L11 -> (arr[0] & arr[1] & arr[2] & !arr[3] & !arr[4]));

-- Sooner or later, the value of i and j will remain equal to 4 forever
LTLSPEC F G (i = 4 & j = 4);

-- Invariant: j is never smaller than i
INVARSPEC i <= j;
```