Embedded Systems

Duration : 3h **All documents allowed**

The three parts are independent. Please answer on 3 separate sheets.
Informal explanations in plain english will be appreciated a lot, and it is compulsory to justify all answers.
The number of points associated with each question is only an indication and might be changed slightly.

Part I - Modeling Time and Concurrency (6 points)

Consider a device R having three operating modes `Idle`, `Transmit` and `Receive`. The initial mode is `Idle`. The commands to be used for requesting a mode change, and the time it takes to change modes, are given below (time is expressed in some time unit which does not need to be specified physically here). If the device receives a command `x2y` when it is not in mode `x`, it is considered as an incorrect use.

Mode change in R	command	change time
Idle to Transmit	`i2t`	3
Idle to Receive	`i2r`	4
Transmit to Receive	`t2r`	5
Receive to Transmit	`r2t`	10
Transmit to Idle	`t2i`	2
Receive to Idle	`r2i`	2

▷ **Question 1 (1 point) :**

Represent the behavior of this device by an automaton AR with inputs `i2t, i2r, t2r, r2t, t2i, r2i` and `top`, where `top` represents an external clock that delivers the units of time (you do not need to draw all the states precisely, but explain the structure of the automaton carefully). To simplify the construction, we can consider that `top` never occurs together with a command in { `i2t, i2r, t2r, r2t, t2i, r2i` }, and that there cannot be two such commands at the same time.

Now we consider a low-level program using this device. It is represented by the Mealy automaton AP of Figure 1, where the inputs `A, B, ...` can be considered as calls from some upper layer, and the outputs correspond to the commands sent to the device R. The input `top` of AP also serves for counting time, on the same scale as AR. We use a simplified notation *else* which means: any input condition for which there is no explicit transition from the state. For instance, from state γ, the *else* covers all the calls `A, B, ...` from the upper layer. From δ, the *else* covers `top`, and the calls other than `B`.

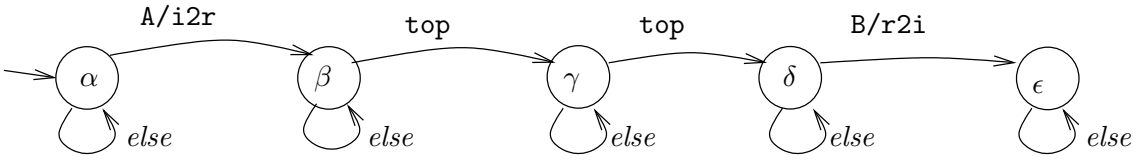

Figure 1: A low-level program AP using the device

▷ **Question 2 (1 point) :**

Explain what happens to the device when the low-level program takes the path $\alpha, \beta, \gamma, \delta, \epsilon$. Explain how to modify the program in such a way that this does not happen.

If we compute the synchronous product of your AR (from question 1) and the AP of Figure 1, and then perform the encapsulation by the signals `i2t, i2r, t2r, r2t, t2i, r2i`, what we obtain is an automaton with inputs `A, B, ...` and `top`, representing the joint behavior of the device and the low-level program.

▷ **Question 3 (1.5 points) :**

We would like to detect the kind of situation described in question 2, for instance with model-checking tools. Explain how to use error states in your AR from question 1, in such a way that the product of AR with AP now reaches an error state whenever a situation like the one of question 2 occurs.

We now consider a variant R' of the device R, which is able to send an interrupt when it has reached a mode. The program can request a mode change by sending a command `x2y`, and it will receive an input `yok` when the mode `y` is entered.

▷ **Question 4 (2.5 points) :**

— In a program using R', it is no longer necessary to count time. Give an example of a program that uses the device R' correctly (in the form of a Mealy automaton, like the example of Figure 1).
— Of course, it is still possible to make a mistake in a program using R'. Give an example of a program that uses R' incorrectly.
— Explain how to add error states to the model of R', in such a way that these incorrect uses of R' can be detected by model-checking.

Bonus question (will only give more points)

▷ **Question 5 (2 points) :**

For the models described above, can you implement a tool that detects incorrect uses of the device automatically, in an exact way? or is it "too complicated", so that only approximate answers can be obtained automatically? Justify your answer very precisely.

Part II - Abstractions and Abstract Operations (4 points)

We consider a special data structure `D`, defined as an array of dimension 1, of size n, indexed from 1 to n, containing integer values. Figure 2 gives two example values for a variable `d` of type `D`.

| 4 | 12 | 7 |, | 5 | 0 | -3 |

Figure 2: Example values of type `D`, with $n = 3$

The following operations are defined for a variable `d` of type `D`:

- `Plus1 (d)` adds 1 to all the elements of `d`
- `IsPositive (d, k)` is true if and only if $d[k] > 0$.
  In the first example value above, `IsPositive(d, 3)`.

Consider a set S of values of type `D` (for size n): $S = \{d_i \mid i \in 1..p\}$. The following abstraction is proposed: S is abstracted into a vector v of size n, $v =< v_1, v_2, ..., v_n >$, where each $v_k = \max_{i=1}^{i=p} d_i[k]$.

▷ **Question 6 (1 point) :**

— Give the vector $v =< v_1, v_2, v_3 >$ that abstracts the set of values of Figure 2.
— Define the abstract values $\top$ and $\bot$

▷ **Question 7 (1 point) :**

Consider two abstract values $v =< v_1, ..., v_n >$ and $v' =< v'_1, ...v'_n >$.
— Define the partial order $v \preccurlyeq v'$ expressing that v is more precise than v'. In other words: if $v \preccurlyeq v'$, then the set of values abstracted by v is included in the set of values abstracted by v'.
— Define the union of v and v'

Now, consider a program P that manipulates a variable d of type `D` (size n).

At some point, the program does: `d := Plus1(d)`. If we know that, just before the assignment, the set of possible values of `d` is abstracted by $v =< v_1, ..., v_n >$, we know that just after the assignment, the set of possible values of `d` is abstracted by $v' =< v'_1, ..., v'_n >$.

▷ **Question 8 (1 point) :**

— Define v' in terms of v.

The program P may also use the condition `IsPositive` at some point, in a structure like: `if IsPositive (d,k) then ... else ...`. If we know that, just before the condition, the set of possible values of `d` is abstracted by $v =< v_1, ..., v_n >$, we know that, at the beginning of the `then` (resp. `else`) branch, the set of possible values of `d` is abstracted by $v' =< v'_1, ..., v'_n >$ (resp. $v'' =< v''_1, ..., v''_n >$).

▷ **Question 9 (1 point) :**

— Define v' and v'' in terms of v.

Part III - Model-Checking (11 points)

Exercise 1: Observers - 4 points

As usual, when talking about Boolean flows, we note 0 for false and 1 for true.

A Boolean flow is said to be a *sporadic event* if it is true only once at a time (it is never true two or more instants in a row). For instance 00010100001000100 can be the beginning of a sporadic event, while 011100100 or 10011001 cannot.

▷ **Question 1 (1 points) :**

Write a Lustre observer `node sporadic(x: bool) returns (ok: bool);` such that the output `ok` remains always true if and only if the input `x` is a sporadic event.

Two sporadic events `x` and `y` are *alternating* if the instants where `x` is true alternate with those where `y` is true, and never match. The following time diagram give an example of alternating events:

x	0	1	0	0	0	1	0	0	1	0	0	1	0	0	1
y	0	0	0	0	1	0	1	0	0	0	1	0	1	0	0

▷ **Question 2 (3 points) :**

Write a Lustre observer `node alternate(x,y: bool) returns (ok: bool);` such that the output `ok` remains always true if and only if `x` and `y` are alternating sporadic events. Warning: the first flow to be true can be any of them.

Exercise 2: Serial-to-parallel converter - 7 points

The goal of this exercise is to program in Lustre a small device that converts a sequence of bits (serial transmission) into a sequence of words (parallel transmission). The questions are progressive and their solutions are very important to reach the final goal.

Periodic flows.

▷ **Question 1 (1 point) :**

— Write a node `R0(x: bool) returns (y:bool)` that implements a register initialized to false (i.e. $y_0 = 0$, $y_{t+1} = x_t$)
— Write a node `R1(x: bool) returns (y:bool)` that implements a register initialized to true (i.e. $y_0 = 1$, $y_{t+1} = x_t$)
— Give some values and an informal description of the flow `a = R0(R0(R1(a)))`
— Give some values and an informal description of the flow `b = R1(R0(b))`

Serial transmission with parity bit. Serial transmission consists in sending binary words, bit by bit, starting with the lowest significant bit. In this exercise, the size of the words is fixed to 4.

For transmitting a word $x_3x_2x_1x_0$, a device first transmits bit x_0, then x_1 etc. After sending the 4 significant bits, an extra bit p (called *parity bit*) is transmitted. On the sender side, this bit is computed in such a way that the number of 1's in x_0, x_1, x_2, x_3, p is EVEN. For instance, the sender may transmit 0101 and then 0 as the parity bit. or 0111 and then 1 as the parity bit; Such a sequence of 5 bits (significant + parity) is called a *serial word*.

On the receiver side, this extra bit is used to check errors: if the number of 1's in the received bits x_0, x_1, x_2, x_3, p is actually ODD, some error has certainly occurred during the transmission. Otherwise,

the transmission is considered as ok. For instance, if 01111 is received the transmission is ok, if 01011 is received, the reception has failed. Notice that this method is not complete and cannot detect 2 errors in a single word.

▷ **Question 2 (1 points) :**

Give a **Boolean** expression over the variables `x0, x1, x2, x3, p` which is true if and only if the number of true values is odd.

The receiver. The Lustre header of the serial receiver is the following:

```
node serial2para(b: bool) returns (x0, x1, x2, x3, pok, perr : bool);
```

where:
— the input flow `b` is the serial information (significant bits + parity bit) as defined before,
— the outputs `x0, x1, x2, x3` are the (parallelized) bits of the word,
— the output `perr` is the parity error bit, it is true if and only if a whole serial word has just been received and a parity error has been detected,
— the output `pok` is the acknowledgement bit, it is true if and only if a whole serial word has been received and no parity error has been detected.

The following table gives an example of execution of `serial2para`; note that values of `x0, x1, x2, x3` are irrelevant unless a whole serial word has been read, this is why a simple dot is used (i.e., any value is acceptable).

b	1	1	0	0	0	1	0	0	0	0	0	1	1	1	1
x0	.	.	.	.	1	.	.	.	.	1	.	.	.	.	0
x1	.	.	.	.	1	.	.	.	.	0	.	.	.	.	1
x2	.	.	.	.	0	.	.	.	.	0	.	.	.	.	1
x3	.	.	.	.	0	.	.	.	.	0	.	.	.	.	1
pok	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
perr	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0

▷ **Question 3 (4 points) :**

Write the program `serial2para`. Explain your choices precisely (intermediate variables, reusing of existing nodes, etc.). Once again: integer flows are forbidden, the program must be purely Boolean.

Binary-Coded Decimal.
The Binary-Coded Decimal (BCD) is a method for encoding numbers in digital devices. Just like in the "human world", numbers are decomposed in base 10, but the decimal digits (0 to 9) are themselves encoded in base 2, using a 4-wide binary word. For instance, the decimal digit 0 is encoded by the binary word 0000, 1 by 1000, 3 by 1100, 9 by 1001 (binary words are represented in big-endian form).

This method is costly in terms of space, since some words are never used: 0101 (ten), 1101 (eleven) etc.

▷ **Question 4 (1 point) :**

Augment the program `serial2para` with a new output `bcd` that checks whether the input words are valid BCD-digits; i.e., `bcd` is true if and only if a whole serial word has just been received and is a valid BCD-digit. There is no need to rewrite the whole program, just give the additional code and the explanations.