Embedded Systems

Duration : 3h **All documents allowed**

The two parts are independent. Please answer on 2 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- Synchronous Programming and Model-Checking (11 points)

I.1: Observers and Boolean flows.

In control engineering, it is often convenient to specify that a Boolean flow represents an "event". Intuitively, an event is something that occurs form time to time, but never lasts more than one instant. For instance, the sequence $0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, \cdots$ is (the beginning of) an event, while $0, 0, 1, 1, 0, 1, 1, 1, \cdots$ is not.

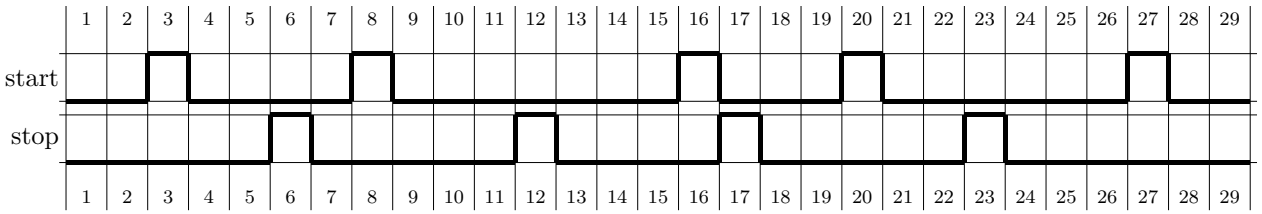
▷ **Question 1 (2 points) :**

Write a Lustre observer, whose header is:

```
node is_event(x: bool) returns (ok: bool);
```

such that `ok` is infinitely true if and only if the input flow `x` is an event. Please explain and comment precisely your solution.

A useful property/assumption when reasoning about events is that two events are alternating over time. This is for instance the case when two signals are supposed to represent respectively the beginning and the end of some treatment. The following timing diagram gives a example of alternating events:



▷ **Question 2 (3 points) :**

Write a Lustre observer, whose header is:

```
node are_alternating(start, stop: bool) returns (ok: bool);
```

such that the output `ok` is infinitely true if and only if the input flows `start` and `stop` are **events** that **alternate** over time. Please explain and comment precisely your solution.

I.2: Model-checking: Symbolic Reachable States computation.

We consider in this exercise deterministic finite state systems, given as Boolean automata. We do not consider here the verification of a particular property, but only focus on the computation of the reachable states of the system. Moreover, in order to simplify, we suppose that we have no hypothesis (assumption). A system is then characterized by:

- a set S of state variables; we note $q \in Q = \mathbb{B}^{|S|}$ a value of the state variables (i.e. a state);
- a set V of free variables (inputs); we note $v \in 2^{|V|}$ a value of the inputs;
- an initial state $q_0 \in Q$; we also note $Init = \{q_0\}$;
- a transition function $T : \mathbb{B}^{|S|} \times \mathbb{B}^{|V|} \to \mathbb{B}^{|S|}$, which associate with each state q and each input value v the corresponding next state; instead of writing $q' = T(q, v)$ we will use the more "visual" notation $q \xrightarrow{v} q'$;

We recall the definition of the $Post(X)$ function, which computes the set of states that can be reached in one transition form a state in X:

$$Post(X) = \{q' \in Q \mid \exists q \in X, \exists v \in V \;\; q \xrightarrow{v} q'\}$$

▷ **Question 3 (2 points) :**

Prove that the *Post* function distributes over set union:

$$\forall X, Y \subseteq Q \;\; Post(X \cup Y) = Post(X) \cup Post(Y)$$

Is it the same for intersection ($Post(X \cap Y) = Post(X) \cap Post(Y)$) ? the complement ($Post(\neg X) = Post(Q \setminus X)$) ? Illustrate your answer with a counter example, if needed.

We saw in the course that the computation of the reachable states consists in computing a sequence of nested sets:

$$A_0 = Init, \quad A_{n+1} = A_n \cup Post(A_n)$$

until we find the first k such that $A_{k+1} = A_k$.

▷ **Question 4 (2 points) :**

To obtain the same result, in the same number of iterations k, is it necessary to compute at each step the *Post* of the **whole set** of states reached so far? If no, give the set with smallest cardinality, B_n, whose *Post* is really necessary at each step n (reminder: cardinality = number of elements).

When sets are encoded with BDDs, the size of the encoding is not directly related to the cardinality. Remember, for instance, that the always-true BDD (size = 1) encodes the whole state space (cardinality = $2^{|S|}$). To avoid any mistake, the size of the BDD encoding X is called the *symbolic size of* X, and is noted $\#X$. The cardinality is noted as usual $|X|$.
The complexity of $Post(X)$ is exponential in the symbolic size of X, In particular, even if $X' \subseteq X$, and thus, $|X'| \leq |X|$, the cost of $Post(X')$ can be exponentially greater than the cost of $Post(X)$. In order to optimize the *Post*, it is then impossible to rely on classical set operations (intersection, union).
There exist operators, specific to the BDD representation, and whose goal is to reduce the size according to the notion of *care-set*. An example of such an operator is the *restrict* operator, noted $\downarrow$. We do not give here the precise definition, but just the properties of interest. Let $Z = X \downarrow Y$:

- X is the main argument, Y is called the care set;
- inside the care set Y, the result Z and the argument X strictly match:
  $\forall q \in Y \;\; (q \in Z) \Leftrightarrow (q \in X)$,
  outside the care set, there is no particular relation between X and Z;
- they are many sets satisfying the constraint above; it is possible to prove that any set Z such that $(X \cap Y) \subseteq Z \subseteq (X \cup \neg Y)$ is a "candidate result". For instance X itself is a candidate, but also $X \cap Y$, which is the solution with minimal cardinality, and $X \cup \neg Y$ wich is the solution with maximal cardinality;
- the goal of the restrict operator is to find a candidate whose symbolic size is *as small as possible*; the result is in general not optimal but we have some guarantee on its quality: it is at least "not worst" than the 3 obvious solutions:
  $\#Z \leq \#X, \quad \#Z \leq \#(X \cap Y), \quad$ and $\#Z \leq \#(X \cup \neg Y)$.

▷ **Question 5 (2 points) :**

Explain how the restrict operator can be used to optimize the computation of the reachable states.

Part II - Timing and the HW/SW Interface (10 points)

In this part, for all the questions below, we consider a piece of hardware, called HW, which offers several operations $\texttt{op}_i$s, and a piece of sequential software, called SW, that uses them. HW has a set of instructions for use, given as an automaton (because not all sequences of operations are allowed). Moreover, we will consider two cases: (i) the operations of HW do not take time, and the calls from SW are synchronous; (ii) the operations do take time, the calls from SW are asynchronous, based on the information "finished" from HW to SW. Finally, we will look at WCET for some examples of SW using HW, based on the timing of the $\texttt{op}_i$s.

II.1: Specification of HWu (untimed operations): We consider a version HWu of HW, in which the operations do not take time. HWu has three operations `op1, op2, op3`. `op1` is used as a kind of initialization, to put HWu in a clean state; it should be used befiore anything else can be done; it can be used at any time to re-initialize HWu to a clean state. After initialization (or re-initialization), the operations `op2` and `op3` can be used, alternating, starting with `op2`.

▷ **Question 6 (1 point) :**
Give a Mealy machine `MHWu` to describe the constraints on the use of operations as explained above. The $\texttt{op}_i$s are the inputs. Explain how you specify incorrect uses.

Now we consider an example SW, named SW1 (see figure 1).

▷ **Question 7 (1 point) :**
Does SW1 make a correct use of HWu (as specified above)? If the answer is no, propose a simple modification of SW1 such that the answer becomes yes.

▷ **Question 8 (1 point) :**
Give a Mealy machine `MSW1` representing SW1, such that the synchronous product of machines `MHWu` (from question 6) and `MSW1` represents the synchronous calls from SW1 to HWu. How do you detect incorrect uses of HWu by SW1?

```
// SW1
b := false ;
op1;
while true loop
   if b then op2 ;
   else      op3 ;
   b := not b ;
end loop
```

```
// SW2
b := false ;
op1; wait(f);
while true loop
   if b then op2 ;
   else      op3 ;
   wait (f) ;
   b := not b ;
end loop
```

Figure 1: Example software, SW1 and SW2

II.2: Specification of HWt (timed operations): We now consider another version HWt of HW, in which the operations *do* take time. The constraints on the use of operations are the same as in paragraph **II.1** above. Additionally, we know that `op1` take 5 units of time, `op2` takes 2, and `op3` takes 4. When an operation is finished, HWt provides the information `f` (for "finished") to the software (in real life, it can be an interrupt, typically).

We also consider example software, named SW2 (see figure 1) making use of HWt. We assume an operation `wait(f)` representing the fact that the software is blocked at this point until the information `f` arrives.

▷ **Question 9 (1.5 points) :**

— Give a Mealy machine `MHWt` to describe the constraints and the timing of operations of HWt (as explained above). Explain what are the inputs and outputs, and how you represent time.
— Give a Mealy machine `MSW2` representing SW2, such that the synchronous product of machines `MHWt` and `MSW2` represents the asynchronous calls from SW2 to HWt. How do you specify and detect incorrect uses of HWt by SW2?

▷ **Question 10 (1.5 points) :**

Same question as the previous one, but with the timed automata of Uppaal: Give a timed automaton to represent HWt, and another one to represent SW2, such that their composition represents the asynchronous calls from SW2 to HWt. Justify your answers.

II.3: We now look at WCET computation. We focus on the timing of operations (`op1` take 5 units of time, `op2` takes 2, and `op3` takes 4), and we neglect everything else (like the timing of the operations on the Boolean variables).

▷ **Question 11 (2 points) :**

— What's the WCET of SW3 (Figure 2)? Explain carefully how you compute it. Do you get a precise evaluation of the WCET?
— Same questions for SW4.

```
// SW3
b := true ;
op1; wait(f);
for i in (1..4) loop
   if b then op2 ;
   else      op3 ;
   wait (f) ;
   b := not b ;
end loop
```

```
// SW4
op1; wait(f);
for i in (1..4) loop
   get (b) // read a value from keyboard
   if b then op2 ;
   else      op3 ;
   wait (f) ;
end loop
```

Figure 2: Example software, SW3 and SW4

Finally, we look at versions of HW for which the timing of operations depends on the state. Consider a version HWs of HW for which: `op1` always take 5 units of time, `op2` always takes 2, and for `op3` it depends: the first occurrence takes 3 units of time, the second one takes 5, the third one takes 3 units again, the fourth takes 5, etc.

▷ **Question 12 (2 points) :**

— What's the WCET of SW3 (Figure 2), for this new HWs? Explain carefully how you compute it. Do you get a precise evaluation of the WCET?
— Same questions for SW4 (Figure 2).