Embedded Systems

Synchronous Approach

Pascal Raymond

Duration : 3h **All documents allowed**

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.

Exercice 1 - Esterel, Automata and Lustre

This exercise is freely inspired by a mouse system for detecting simple or double clicks. This system receives two signal: `c` when the mouse button is pressed, `t` which is a tick signal coming from some system clock. The behavior is the following: each time `c` is pressed, if `c` is pressed again **before** two occurrences of `t`, the system decides that it is a double-click and emit the output `D`, on the contrary, if no `c` has been received during this time the system decides that it is a simple click and emit `S`.

This behavior is implemented by the following Esterel program:

```
module CLICKS:
input c, t;
output S, D;
present c else
   await c
end;
loop
   abort
      await t;
      await t;
      emit S
   when c do
      emit D
   end;
   await c
end.
```

▷ **Question 1 (3 points) :**

Following the method viewed in course, build the Mealy machine (finite automaton) equivalent to this Esterel program. We remind that the main steps for that are: – identify the states (control points), – find all the possible control paths between two states (transitions), together with the corresponding condition/emission.

The goal is now to implement exactly the same specification using the Lustre language. For that, we will use the natural correspondence between events (i.e. signal) and Boolean: present = true, absent = false.

▷ **Question 2 (3 points) :**

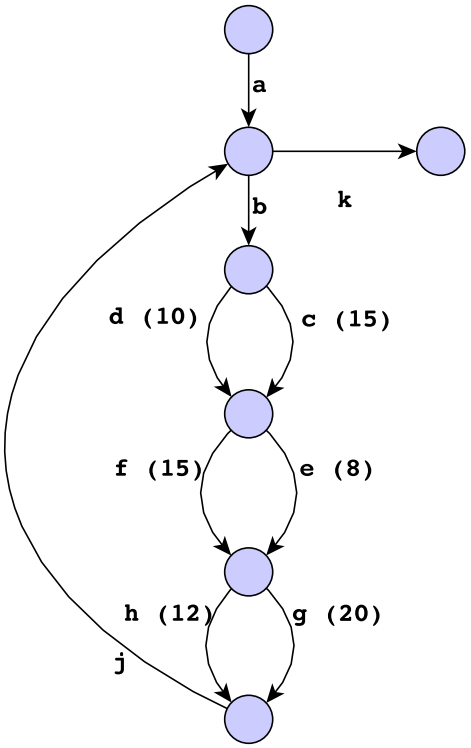
Write a Lustre program that implements the CLICKS specification, and whose header is:

```
node CLICKS(c, t: bool) returns (S, D: bool);
```

Exercice 2 - WCET Estimation and Infeasible paths

```
const int N = 10;

void foo(int x[N]){
   int i,y,z;
   // a
   for (i=0; i < N; i++) {
      y = x[i]; // b
      if (read()) {
         z = 0; // c:  15 cycles
      } else {
         z = (); // d:  10 cycles
      }
      if(read()) {
         y = read(); // e:  8 cycles
      } else {
         y += z; // f:  15 cycles
      }
      if(y > x[i]) {
         ...  // g:  20 cycles
      } else {
         ...  // h:  12 cycles
      }
      ...  // j
   }
   ...  // k
}
```



The figure above shows, **on the right hand side**, the Control Flow Graph (CFG) of a program for which we want to estimate the Worst Case Execution Time (WCET). A micro-architecture analysis tool has been used to estimate the execution time of each basic block and each transition. In order to simplify, these local weights have been distributed to some edges only: **c** and `f` cost 15 cpu cycles, **d** costs 10, **e** costs 8, **g** costs 20 and **h** costs 12. *We suppose that all other edges cost 0.*

▷ **Question 3 (1 point) :**

Encode the Worst Path (WP) problem into a first Integer Linear Program, containing:
— the set of structural constraints imposed by the CFG,
— the objective function (global WCET) to maximize.

Without any further information, the WCET obtained from this first ILP is obviously infinite. In order to bound the execution paths, it is necessary to analyse the *semantics* of the program. The **left-hand side** of the figure shows the simplified code of the program, in C-like syntax. The comments give the correspondence between the code and the CFG transitions. The code is simplified and shows only some variables of interest. These variables (`x,i,y,z`) are accessed and modified ONLY when mentioned. Moreover there are no information on the function `read()`, and its result should be considered as random.

▷ **Question 4 (2 points) :**

— Deduce from the program code an upper bound of the number of loop iterations.
— Give a corresponding integer constraints to add to the ILP problem.
— Give a first (finite) estimation of the WCET.

We will now try to enhance the WCET estimation by searching infeasible paths in the CFG, and remove them from the WP search using suitable extra linear constraints.

▷ **Question 5 (4 points) :**

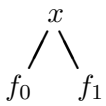
Does it exist execution paths that are (provably) infeasible in this program ? (**Justify precisely your answer**). If it is the case:
— Express these infeasibility with a numerical constraint.
— Give an (new) estimation that take this constraint into account.

Exercice 3 - Extracting implicants from BDDs

In this exercise, we consider Boolean formulas over a finite set of ordered variables, together with the corresponding representation with Binary Decision Diagrams (BDDs).

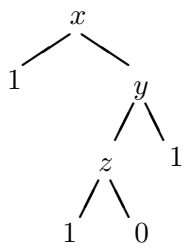
Remarks: *"+" denotes logical or, "·" logical and, "¬" negation (or "$\bar{x}$" for a single variable), "⊕" exclusive or, "⇒" implication. For the sake of simplicity and readability, BDDs are always represented by* **trees**, *i.e. without the sharing of common sub-graphs.*

We adopt the same convention as in the course: in a binary node labeled with a variable x, the right branch corresponds to the case "x is true", and the left branch to the case "x is false:



$$\Leftrightarrow \quad x \cdot f_1 + \bar{x} \cdot f_0$$

For instance, the BDD (more precisely the Shanon Reduced Tree) of the formula $x \Rightarrow (y + \bar{z})$ is:



▷ **Question 6 (1 point) :**

Give (draw) the BDD of the formula $y \oplus (x \cdot z)$ for the variable order $x < y < z$.

Let's recall some basics of Boolean algebra:

— A **literal** is a formula reduced to one variable (e.g. x) or the negation of one variable (e.g. $\bar{x}$).
— A **monomial** is a conjunction of literals where each variable appears at most once (e.g. $x \cdot \bar{y} \cdot z$).
— An **implicant** of a formula f is a monomial m such that $m \Rightarrow f$. For instance, $\bar{x}$ and $x \cdot \bar{z}$ are implicants of $x \Rightarrow (y + \bar{z})$.
— A **prime implicant** of a formula f is an implicant such that no other implicant $m' \neq m$ satisfies $m \Rightarrow m'$. In other terms, a prime implicant is a "shortest" sufficient condition to satisfy f.
For instance, $\bar{x}$ is a prime implicant of $x \Rightarrow (y + \bar{z})$, but $x \cdot \bar{z}$ is not prime because it implies $\bar{z}$ which is itself an implicant of $x \Rightarrow (y + \bar{z})$.

▷ **Question 7 (2 points) :**

Give **all** the implicants of $y \oplus (x \cdot z)$, Indicate which of them are prime.

Finding implicants is very important, in particular for debugging and diagnosis purpose: a prime implicant is a "minimal" explanation of why a formula is satisfiable. Unfortunately, the search of

prime implicants is known to be very costly (NP-hard). The practice is to find "small" implicants, with a reduced number of variables, but not guaranteed to be prime.

The goal of this exercise is to define such an algorithm, named $\mathcal{I}$, that takes advantage of the Shanon decomposition. This algorithm:

— takes as input a BDD which is **neither 0** (that has trivially no implicant), **nor 1** (for which any monomial is an implicant),
— returns **one implicant**, not necessarily prime, but minimized.

Suppose that the BDD of f is of the form: $\begin{array}{ccc} & x & \\ / & & \backslash \\ f_0 & & 0 \end{array}$, it means that $f = \bar{x} \cdot f_0$ and thus, the literal $\bar{x}$ **must** appear in any implicant of f. A reasonable rule for computing $\mathcal{I}$ in this case is then:

$$\mathcal{I}\left(\begin{array}{ccc} & x & \\ / & & \backslash \\ f_0 & & 0 \end{array}\right) = \bar{x} \cdot \mathcal{I}(f_0)$$

By using this kind of reasoning, we propose to define a $\mathcal{I}$ algorithm by giving a (recursive) rule for all the possible cases. To forbid any ambiguity and priority problems, we have identified 7 exclusive cases: exactly one of them is applicable to a BDD different to 0 and 1.

▷ **Question 8 (4 points) :**

Propose a $\mathcal{I}$ algorithm by giving a rule for all the following cases, where f_0 and f_1 denote BDDs different from 0 and 1:

$$\mathcal{I}\left(\begin{array}{ccc} & x & \\ / & & \backslash \\ 0 & & 1 \end{array}\right) =? \quad \mathcal{I}\left(\begin{array}{ccc} & x & \\ / & & \backslash \\ 1 & & 0 \end{array}\right) =? \quad \mathcal{I}\left(\begin{array}{ccc} & x & \\ / & & \backslash \\ f_0 & & 1 \end{array}\right) =? \quad \mathcal{I}\left(\begin{array}{ccc} & x & \\ / & & \backslash \\ 1 & & f_1 \end{array}\right) =?$$

$$\mathcal{I}\left(\begin{array}{ccc} & x & \\ / & & \backslash \\ 0 & & f_1 \end{array}\right) =? \quad \mathcal{I}\left(\begin{array}{ccc} & x & \\ / & & \backslash \\ f_0 & & 0 \end{array}\right) =? \quad \mathcal{I}\left(\begin{array}{ccc} & x & \\ / & & \backslash \\ f_0 & & f_1 \end{array}\right) =?$$

NOTE: Explain precisely your reasoning. In particular, whenever there are several solutions for extracting an implicant, **say it, and explain why** you have chosen one solution rather than another.