Embedded Systems

Arnaud Legrand, Pascal Raymond

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.

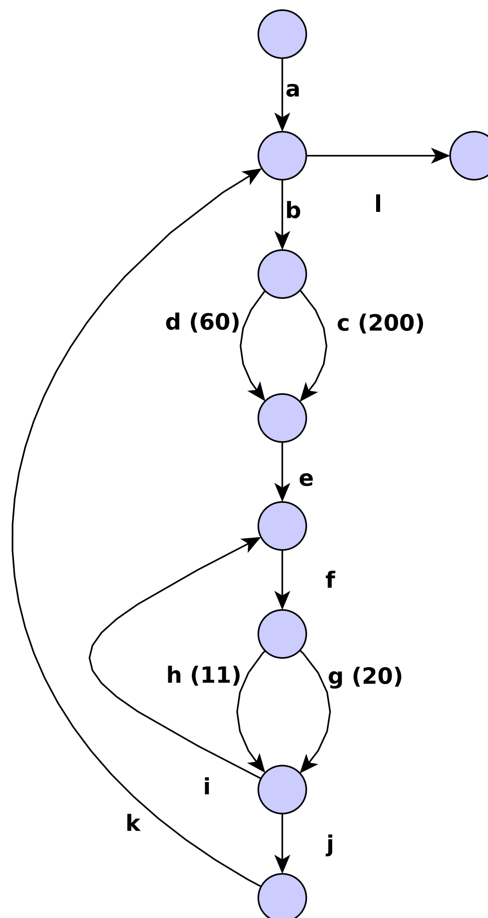
%cleardoublepage

Part II - Synchronous Approach

Exercice 1 - WCET Estimation and Infeasible paths

```
const int H = 5;
const int W = 10;

void foo(){
  int x,y;
  int cond, val;
  /*a*/
  for(x=0; x < H; x++){
    /*b*/
    cond = read();
    if (cond > 0) {
      /*c: 200 cycles*/
      val = 0;
    } else {
      /*d: 60 cycles*/
      val = read();
    }
    /*e*/
    for(y=0; y < W; y++){
      /* f */
      if (val > 0){
        /*g: 20 cycles*/
      } else {
        /*h: 11 cycles*/
      }
       /*i*/
    }
    /*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). This CFG has 9 basic blocks and 12 edges, identified with the letters from **a** to **l**. 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** costs 200 cpu cycles, **d** costs 60, **g** costs 20, **h** costs 11. *We suppose that all other edges cost 0.*

▷ **Question II.1 (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 represents a simplified version of code of the program, in C-like syntax. The comments give the correspondence between the code and the CFG branches. The program variables are modified ONLY when mentioned in the simplified code. In particular, `val` is modified only at the branches **c** and **d**, everywhere else it keeps its current value. ▷ **Question II.2 (2 points) :**

— By considering the code, find bounds for each loop in the program. **Explain as precisely as possible how you obtain these bounds.**
— Give the corresponding integer constraints to add to the ILP problem.
— Give a first (finite) estimation of the WCET.

We can 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 II.3 (2 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
— Explain why this new estimation is or is not better than the first one.

Exercice 2 - Lustre Programming

Periodic Flows in Lustre

▷ **Question II.4 (2 points) :**

— Write a Lustre node whose header is:

```
node R0(x:bool) returns (y:bool);
```

and which implements a *false-inititialized* register, i.e. such that $y_0 = false$ and $y_{t+1} = x_t$

— Write a Lustre node whose header is:

```
node R1(x:bool) returns (y:bool);
```

and which implements a *true-inititialized* register, i.e. such that $y_0 = true$ and $y_{t+1} = x_t$

— Consider the Lustre *recursive* equation `a = R0(R0(R1(a)));`

- give the first 7 or 8 values of the flow `a`
- give an informal description of this flow `a`

Serial Transmission and Parity Bit

We consider the *serial* transmission (i.e. one bit after one bit) of elementary words. For the sake of simplicity we consider elementary words made of 3 bits only (in reality, elementary data are indeed bytes of 8 bits). In serial transmission, it is common to complete the transmission of one word with an extra bit, *called parity bit*, that helps to check whether the transmission was correct or not:

— let x_0, x_1 and x_2 be the 3 bits of the transmitted word, the parity bit p should be 0 (false) if the number of 1's in x_0, x_1, x_2 is even, and should be 1 (true) if the number of 1's in x_0, x_1, x_2 is odd.

— for instance, if $x = 000$ or $x = 110$ or $x = 101$ etc. the parity bit should be $p = 0$,

— for instance, if $x = 100$ or $x = 010$ or $x = 111$ etc. the parity bit should be $p = 1$.

▷ **Question II.5 (3 points) :**

Give a logical expression over the variables x_0, x_1, x_2 and p which is true if and only if p is the correct parity bit for the word x_0, x_1, x_2.

We now want to define an observer that take as input a Boolean flow which is supposed to be the (unbounded) serial transmission of a word + parity bit flow, and produces an oracle `ok` that must remain true as long as this flow corresponds to a *correct transmission*. For that, the input flow should be interpreted as a sequence of groups, each group being made of 3 data bits followed by the corresponding parity bit.

Here is an example of execution of this observer for a particular sequence i:

	group 1				group 2				group 3				group 4			
	data			par	data			par	data			par	data			par
i	0	0	1	1	1	0	1	0	1	1	1	1	1	1	0	1
ok	*1*	*1*	*1*	**1**	*1*	*1*	*1*	**1**	*1*	*1*	*1*	**1**	*1*	*1*	*1*	**0**

Note that:

— depending on the instant, the input `i` corresponds either to a data bit (instants 0, 1 or 2 modulo 4), or a parity bit (instant 3 modulo 4);

— the value of `ok` may change only at the instants where a parity bit is expected;

▷ **Question II.6 (2 points) :**

Write, in Lustre, such an observer whose header is:

node check_serial(i: bool) returns (ok: bool);