Embedded Systems

Florence Maraninchi, 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.

Part I - Exercice 1 - BDDs and Symbolic Model-Checking (6 points)

I.1: Implementing if-then-else with BDDs

Reminder: let V be a set of Boolean variables, totally ordered by the relation $\preceq$. The BDDs over V, together with their range function rg from BDD to $V \cup \{\infty\}$, are defined inductively by the rules:
— the leaf 1 is a BDD with $rg(1) = \infty$, which denotes the always-true function,
— the leaf 0 is a BDD with $rg(0) = \infty$, which denotes the always-false function,
— the binary node $f = \begin{array}{c} x \\ / \quad \backslash \\ f_0 \quad\quad f_1 \end{array}$, where $x \prec rg(f_0)$ and $x \prec rg(f_1)$, is a BDD with $rg(f) = x$, which denotes the function $x \cdot f_1 + \bar{x} \cdot f_0$.

As we saw in the course, the classical Boolean operations (e.g. **not**, **and**, **or**) can be inductively defined on BDDs. The goal of this exercise is to define the "if-then-else" operation, i.e. a function with 3 BDDs arguments, $\mathbf{ite}(c, f, g)$, that computes the BDD of the function: "if c then f else g".

Is it possible to define **ite** in terms of existing operations:

$$\mathbf{ite}(c, f, g) = \mathbf{or}(\mathbf{and}(c, f), \mathbf{and}(\mathbf{not}(c), g))$$

but we want here to define a more efficient implementation, that exploits the properties of "if-then-else" to reach the result as quickly as possible. Here is for instance a terminal rule that obviously speeds-up the computation:
— $\mathbf{ite}(c, f, f) = f$

However, it is sometimes a good idea to re-use existing operations, for instance:
— $\mathbf{ite}(c, f, 0) = \mathbf{and}(c, f)$

▷ **Question 1 (2 points) :**

Propose a set of terminal rules for the **ite** operation. This set must:
be *sufficient* to ensure the termination of the algorithm,
cover all the cases that can speed-up the computation.

The goal is now to define the recursive rules for **ite**. First of all, we have to check and prove, that, just like for the binary operators, such a recursive descent is *correct*.

▷ **Question 2 (1 point) :**

Prove that the **ite** operation distributes on the Shannon decomposition, that is, if the 3 arguments are binary nodes labeled with *the same variable* x:

$$\mathbf{ite}\left(\begin{array}{c} x \\ / \quad \backslash \\ c_0 \quad\quad c_1 \end{array} , \begin{array}{c} x \\ / \quad \backslash \\ f_0 \quad\quad f_1 \end{array} , \begin{array}{c} x \\ / \quad \backslash \\ g_0 \quad\quad g_1 \end{array} \right) = \begin{array}{c} x \\ / \quad \backslash \\ \mathbf{ite}(c_0, f_0, g_0) \quad\quad \mathbf{ite}(c_1, f_1, g_1) \end{array}$$

It remains now to define the general recursive case, that is, the case where all the variables are not necessarily the same.

Listing one by one the possible *balance rules* is not reasonable since there are plenty of possible orderings for the variables x, y, z. To ease the definition, we assume the availability of the following functions:

— $\mathbf{min}(x, y, \cdots)$ that returns the minimal variable in the list,
— two functions $\mathbf{high} : V \times BDD \mapsto BDD$ and $\mathbf{low} : V \times BDD \mapsto BDD$, such that:

$$\mathbf{high}\left(x,\ \begin{array}{ccc} & x & \\ f_0 & & f_1 \end{array}\right) = f_1 \text{ and } \mathbf{high}\left(x,\ \begin{array}{ccc} & y & \\ f_0 & & f_1 \end{array}\right) = \begin{array}{ccc} & y & \\ f_0 & & f_1 \end{array} \text{ if } x \neq y,$$

$$\mathbf{low}\left(x,\ \begin{array}{ccc} & x & \\ f_0 & & f_1 \end{array}\right) = f_0 \text{ and } \mathbf{low}\left(x,\ \begin{array}{ccc} & y & \\ f_0 & & f_1 \end{array}\right) = \begin{array}{ccc} & y & \\ f_0 & & f_1 \end{array} \text{ if } x \neq y,$$

▷ **Question 3 (1 point) :**

Define the recursive rule(s) for the **ite** operation, in the general case where all arguments are binary nodes:

$$\mathbf{ite}\left(\begin{array}{ccc} & x & \\ c_0 & & c_1 \end{array},\ \begin{array}{ccc} & y & \\ f_0 & & f_1 \end{array},\ \begin{array}{ccc} & z & \\ g_0 & & g_1 \end{array}\right)$$

I.2: Symbolic computation of pre-condidition

We recall that a Boolean state machine is characterized (among other things) by:

— a set of state variable S,
— a set of free variables (inputs) V,
— a vector of transition functions, one per state variables $s \in S$, such that $g_s : \mathbb{B}^{|S|} \times \mathbb{B}^{|V|}$ defines the next value of s according to the current values of the state and input variables.

We consider here the symbolic manipulation of state machines using:

— BDDs over the variables S for representing sets of states (i.e. functions in $\mathbb{B}^{|S|} \to \mathbb{B}$),
— BDDs over the variables $S \cup V$ for representing sets of transitions (i.e. functions in $\mathbb{B}^{|S|} \times \mathbb{B}^{|V|} \to \mathbb{B}$); this is in particular the case of the transition functions.

For the sake of simplicity, we suppose that the transition functions are stored in an array indexed by the state variables: for all $s \in S$, $G[s]$ is the BDD of the transition function g_s.

The pre-condition (also called the *reverse image*) of a set of states X is, by definition, the set of transitions (pairs in $S \times V$) that lead to some state in X.

The goal is here to implement the pre-condition as an operation from BDDs over S to BDDs over $S \cup V$.

▷ **Question 4 (1 point) :**

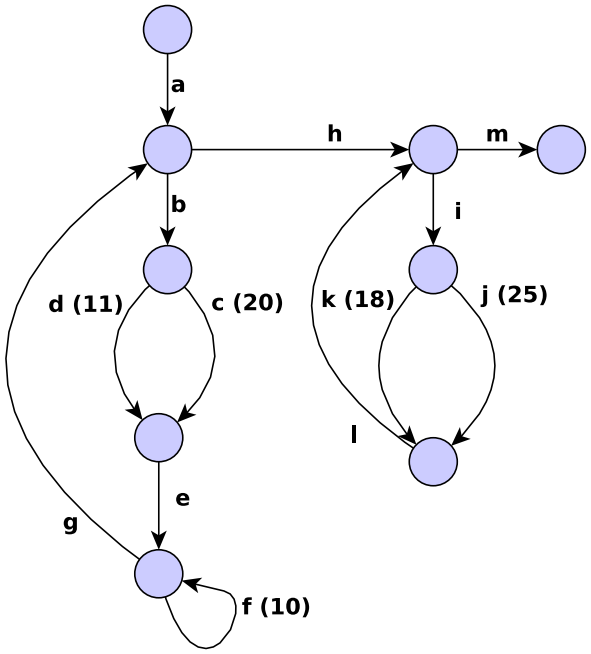
Prove that (or explain as precisely as possible why) $\text{Precond}(1) = 1$
i.e. the precondition of the whole state set is the whole transition set, or equivalently, any transition is the pre-condition of some state.

▷ **Question 5 (1 point) :**

Propose a recursive implementation of $\text{Precond}(X)$ where the argument X is a BDD X over S, and the result is a BDD over $S \cup V$.

Part I - Exercice 1 - WCET estimation and ILP technique (5 points)

```
const int sz = 20;
void foo(int A[sz]){
  int i,j;
  int B[sz];
  /*a*/
  for(i=0; i < sz; i++){
    /*b*/
    if (A[i] < 0) {
      /*c*/
      B[i] = 0;
    } else {
      /*d*/
      B[i] = 2 * A[i];
    }
    /*e*/
    for(j=0; j <= i; j++){
      /*f*/
    }
    /*g*/
  }
  /*h*/
  for(i=sz-1; i >= 0; i--){
    /* i */
    if (B[i] > 0){
      /*j*/
    } else {
      /*k*/
    }
    /*l*/
  }
  /*m*/
}
```



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 13 edges, identified with the letters from **a** to **m**. 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 20, **d** costs 11, **f** costs 10, **j** costs 25 and **k** costs 18. *We suppose that all other edges cost 0.*

▷ **Question 6 (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 the code of the program. In order to be readable, the code is given in C rather than in assembly language. The correspondence between the C code and the CFG is given in comments: for instance, edge **a** is the entry edge, just before the first loop statement; **c** corresponds to the *then* branch of the first *if* statement, etc.

▷ **Question 7 (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 8 (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 one (or more) numerical constraints,
— Are this/these constraint(s) leading to an enhanced estimation? and if this is the case, give a new (enhanced) estimation of the WCET.

Part II - Exercice 1 - Modeling Time (4 points)

In the course we studied a paper about the modeling of a MAC protocol in sensor networks. In the software of a node, time is given by the hardware delivering "ticks" of a discrete clock. There is one such "clock" for each node of the sensor network, and they cannot be considered to be synchronized.

In this exercice, we consider discrete-time synchronous models for this type of problem.

We consider two nodes N_1 and N_2 of the network. Each N_i has an internal hardware device that delivers a sequence of ticks t_i. We also consider a finer grain base clock with ticks t_0. The base clock is precise enough so that each t_1 and each t_2 is also a t_0. Fig. 1 illustrates the idea, for two hardware clocks that should tick each 4 ticks of the base clock, but have an imperfect behavior so that they tick between 3 and 5 ticks of the base clock.

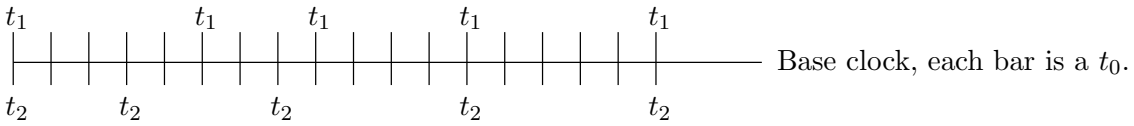


Figure 1: Base clock (t_0) and example sequences of ticks t_1 and t_2 from hardware clocks.

▷ **Question 9 (1 point) :**

— Draw a picture similar to that of Figure 1, illustrating the worst desynchronisation case between t_1 and t_2.
— Give two Mealy machines M_1 (resp. M_2) with input t_0 and output t_1 (resp. t_2); these machines produce t_1 and t_2 from t_0, in the worst case mentioned previously. Note: use the simple form for Mealy machines, with no additional variables.

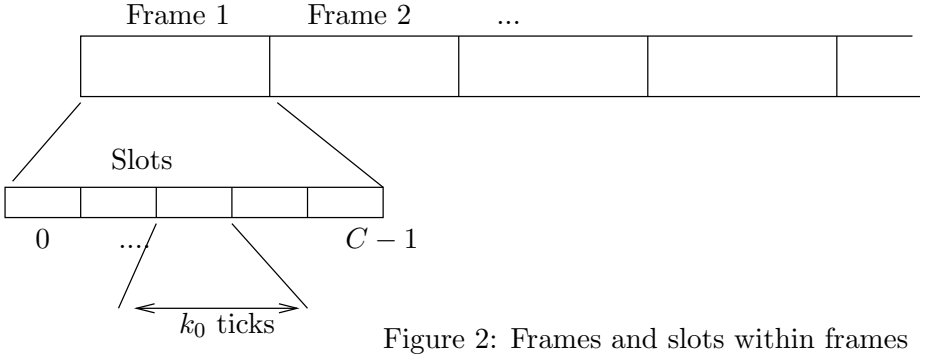


Figure 2: Frames and slots within frames

The protocol described in the paper is based on the notions of *time frame* and *slot*, as shown on Figure 2. Each time frame contains C slots, and a slot has a duration of k_0 ticks of the node.

▷ **Question 10 (1 point) :**

We decide that $k_0 = 10$ and $C = 5$. Give a Mealy machine that computes the slot number in node N_1. You should use a variable `sn1`, initialized to 0, to represent the slot number.
What do you need to change to get the slot number in node 2?

▷ **Question 11 (2 points) :**

— Explain how to compose the Mealy machines of questions 9 and 10 to obtain a new machine G: G has a single input t_0, and computes the slot numbers in the two nodes.
— Using G, how could you express and verify the property: "*if the hardware clocks are not synchronized, the two nodes do not always agree on their slot number*"?

Part II - Exercice 2 - WCET with Simple Caches (6 points)

In the course we studied a paper on WCET and caches. We use the same notions here.

Let us consider a program, given as an *interpreted automaton*. An interpreted automaton is a tuple $(Q, q_0 \in Q, V, T \subseteq Q \times \mathcal{L}(V), Q)$ where Q is the finite set of states (or control points), q_0 is the initial state, V is a finite set of integer variables manipulated by the program, and T is the set of transitions. A transition (q, ℓ, q') from q to q' is labeled by $\ell \in \mathcal{L}(V)$, which can be: either an assignement to a variable in V (e.g., `x:= x+1` or `x:= y+z`), or a test on variables in V (e.g., `x < y` or `x > 42`). We will denote by $\mathcal{V}(\ell)$ the set of variables that appear in a label ℓ. For instance $\mathcal{V}($`x := y+z`$) = \{$ `x, y, z` $\}$.

The architecture on which the program is executed has a main memory (an array of m blocks, indexed from 1 to m). Each variable is installed somewhere in the main memory, once and for all. We note $b($`x`$)$ the index of the block in which `x` resides.

The architecture on which the program is executed also has a LRU cache, which is an array of n blocks, indexed from 1 to n. The cache is initially empty. At any moment during the execution, a concrete state of the cache is a function c from $[1..n]$ to $[1..m] \cup \{I\}$. $c(k) = h$ means that the block k of the cache currently contains the block h of the memory. $c(k) = I$ means the block k of the cache is currently empty. The set of all concrete cache states is noted $\mathcal{C}$. Each access to a variable `x` of the program updates the concrete state of the cache. The update function is noted $\mathcal{U} : \mathcal{C} \times V \longrightarrow \mathcal{C}$, and an update is noted: $\mathcal{U}(c,$ `x`$) = c'$. Together, $\mathcal{C}$ and $\mathcal{U}$ constitute the definition of an automaton where each transition is labeled by a variable in V.

▷ **Question 12 (1 point) :**

Consider a program state $q \in Q$, and a cache state $c \in \mathcal{C}$. Explain how you compute the new state c' of the cache obtained when the program takes a transition (q, ℓ, q'). If you need to make additional hypotheses, justify them carefully.

We now build an automaton A that represents the behavior of the program together with the cache. It is a kind of synchronous product between: (i) the interpreted automaton of the program; (ii) the cache automaton defined by $\mathcal{C}$ and $\mathcal{U}$.

▷ **Question 13 (2 points) :**

— What is the set of states S of A? What is the initial state s_0 in A?
— Describe the transitions of A. **Indication:** From a state $s \in S$, consider each possible move of the program, and the moves it provokes in the cache (this was question 12); together these moves lead to a new state $s' \in S$; define s' precisely.

We now look at the WCET problem. We consider programs *without loops*, and we decide to count only the time taken by the memory accesses, ignoring all other instructions. One access to `x` costs t if the variable is in the cache, and T if it is not in the cache (there are no timing anomalies).

As an example, we consider the program: `if (x == 0) { y=1; } else { z=1; } ; y=2; z=3;`
We assume that the three variables reside in 3 different blocks of the main memory, and that the cache has $n = 3$ blocks, initially empty.

▷ **Question 14 (2 points) :**

— What is the exact execution time of this program (counting only memory accesses)?
— Draw the interpreted automaton of the example program
— Draw the product A of this program with the cache
— Explain how to use A to compute the WCET of the program.

▷ **Question 15 (1 point) :**

In general, compare the results you can get by using A, and the results described in the paper we studied (precision, cost of the algorithm).