Systèmes Synchrones
MPRI – cours 2.23-1 2016-2017

Final Exam

November 29, 2016

This text has 4 pages. The time limit is 3h. Courses notes are allowed.

Exercice 1

Let $x = (x_n)_{n \in IN}$ and $y = (y_n)_{n \in IN}$ two sequences. $x =_n y$ means that they are equal up to instant $n \in IN$, that is, for all $0 \leq i \leq n$, $x_i = y_i$. A function f is one-to-one synchronous if for all $n \in IN$, $x =_n y \Rightarrow f(x) =_n f(y)$.

Let $\texttt{bool}^{IN}$ the set of sequences of boolean values. Give an example of a function $f : \texttt{bool}^{IN} \rightarrow \texttt{bool}^{IN}$ that is one-to-one synchronous but whose output at instant n may depend on an unbounded past of its input. *Advice: Give a precisely defined function.*

Exercice 2

Write (in Lustre or Esterel) a function with two boolean input signals `a` and `b` which returns true when `a` and `b` alternate, i.e., it is not possible to have two occurrences of `a` without having one of `b` (and reciprocally).

Two inputs `a` and `b` are said to be *quasi-synchronous* if there is at most 2 occurrences of one between two occurrences of the second. Write a Lustre function which returns true whenever its two inputs are quasi-synchronous.

Problem: Sequential operators

In this problem, you will extend a language kernel similar to Lustre with control structures and study their translation into the kernel. The syntax of the language is given below.

$$
\begin{array}{lcl}
d & ::= & \texttt{node}\ f(p)\ \texttt{returns}\ (q)\ D \\
p, q, r & ::= & x \mid x, ..., x \\
D & ::= & D\ \texttt{and}\ D \mid x = e \mid \texttt{var}\ r\ \texttt{do}\ D\ \texttt{done} \\
e & ::= & v \mid x \mid \texttt{true} \mid \texttt{false} \mid e + e \mid e = e \mid e\ \texttt{and}\ e \mid e\ \texttt{or}\ e \mid \texttt{not}\ e \\
 & & \mid \texttt{pre}\ v\ e \mid \texttt{init}\ v \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e
\end{array}
$$

d is the definition of a node with formal parameters p, result q and body D. p, q and r denote patterns; here lists of variables. D stands for equations of the form $x = e$, with e an expression, parallel compositions of equations, D `and` D and hiding a local variable r (`var` r `do` D `done`). v denotes a value (either integer i or boolean, `true`, `false`). + stands for integer addition; `and` for logical conjunction; `or` for logical disjunction, and; `not` for negation. `pre` v x is the previous value of a signal initialized with value v. `init` v is initially v then false, forever.

We write `preb` e as a shortcut for `pre false` e. We will also simply write `pre`(e) for `pre` -1 e.

Reaction semantics: The reaction semantics for this kernel has been given in the course. We remind the main predicates.

Values:	$v \quad ::= \quad i \mid$ `true` $\mid$ `false`	
Environnement:	$R \quad ::= \quad [v_1/x_1, ..., v_n/x_n] \;\; (\forall k, l, k \neq l \Rightarrow x_k \neq x_l)$	
Composition:	R_1, R_2 tq $Dom(R_1) \cap Dom(R_2) = \emptyset$	
Reaction:	$R \vdash e_1 \xrightarrow{v} e_2 \qquad R \vdash D \xrightarrow{R'} D'$ with $R' \subseteq R$	
Run:	$R.h \vdash D : R'.h'$	
History:	$h \quad ::= \quad \epsilon \mid R.h$	

- $R \vdash e_1 \xrightarrow{v} e_1'$ means that, under the local environment R, the expression e_1 produces the value v and rewrites to e_1'.
- $R \vdash D \xrightarrow{R'} D'$ means that, equation D produces R' and rewrites to D'. For that, we maintain the invariant that D sees the signals that are produced, that is, $R' \subseteq R$.

Question 1 Define the following operations in terms of the kernel language:

1. `until`(x) returns a sequence *ok* that is initially false and that only becomes true as soon as x is true in the strict past. Once *ok* becomes true, it stays true.
2. `unless`(x) returns a sequence *ok* with current value true as soon as x is true. The current value of *ok* is false otherwise. Once *ok* is true, it stays true.
3. Express the initialization operation x `->` y in term of the constructs of the language.

The following questions involve extending the kernel language with new programming constructs by defining the cases of a translation function $Tr(.)$ where $Tr(D)$ takes an equation D and returns another equation D'.

Advice: For this translation, you can program it in OCaml provided that you have properly defined the data type for representing abstract syntax trees.

Activation Condition

The kernel language is now extended with an "activation condition" mechanism. The syntax is given below:

$$D \quad ::= \quad \texttt{activate if } e \texttt{ then } D \texttt{ done} \mid ...$$

Intuitively, in `activate if` e `then` D `done`, the equation D is active only at the instants when e is true. Otherwise, variables from D keep their previous values. For example, the following program defines the sequence: `cpt = -1 -1 42 43 43 43 44 45 45 45 46 47 ...`

```
 activate if cond then cpt = 42 -> pre cpt + 1 done
and
 cond = false -> (preb (false -> not (preb cond)))
```

Question 2 Is the previous program equivalent to the following one? Explain why.

```
    cpt = if cond then 42 -> pre cpt + 1 else pre cpt
and cond = false -> (preb (false -> not (preb cond)))
```

Question 3 Propose an equivalent version that does not use the "activation condition" control structure.

Question 4 Define a translation function $Tr(D)$ which translates D from the extended language into a semantically equivalent equation D' from the kernel language.

Question 5 Propose a sufficient condition on `activate if` e `then` D `done` so that its translation is causally correct, in the Lustre sense.

Question 6 [*] Extend the reaction semantics to deal with this new construct. Prove that your translation is correct.

We now extend the syntax and semantics of activation conditions to allow a default handler to be executed when the boolean condition is false.

$$D \quad ::= \quad \texttt{activate if } e \texttt{ then } D \texttt{ else } D \mid ...$$

For example, the following program:

```
activate if cond then cpt = 42 -> pre cpt - 1
                 else cpt = 45 -> pre cpt + 1 done
and cond = false -> (preb (false -> preb (false -> not (preb cond))))
```

defines the sequence `cpt = 45 46 47 42 41 40 48 49 50 39 38 37 ...` (`pre cpt` denotes a local memory updated only when the code in which it appears is active. The two occurrences of `pre cpt` denote different memories).

Question 7 Give an equivalent definition without using the binary activation condition.

Question 8 Extend the translation function $Tr(.)$ accordingly. You may assume that the sets of non-local variables defined in D_1 and D_2 in `activate if` e `then` D_1 `else` D_2 are the same.

Question 9 Extend the translation function $Tr(.)$ to handle the general situation where the two branches do not necessarily define the same variables.

Question 10 [*] Extend the reaction semantics for this new construct `activate if` e `then` D_1 `else` D_2. Prove that your translation preserves the semantics.

Sequencing Operations

We now introduce sequencing constructs.

$$D \quad ::= \quad \cdots \mid \texttt{do } D \texttt{ until } e \texttt{ then } D \mid \texttt{do } D \texttt{ unless } e \texttt{ then } D$$

`do` D_1 `until` e `then` D_2 gives weak preemption: D_1 is activated up to and including the first instant that the boolean condition e becomes true. The execution of E_2 then starts in the following instant. `do` D_1 `unless` e `then` D_2 gives strong preemption: D_1 is executed up to but not including the first instant that e is true. D_2 starts at the first instant when e is true. Thus, the program:

```
do x = 0 -> pre x + 1 until (x = 5) then x = 10 done
```

defines the sequence `x = 0 1 2 3 4 5 10 10 10 10 ...` The following program:

```
  do x = 0 -> pre x + 1 unless cond then x = 10 done
and
  cond = false -> preb (false -> true)
```

defines the sequence `x = 0 1 10 10 10 ...`

Question 11 Extend the translation function $Tr(.)$ with the two sequencing constructs.

Question 12 Define a causality constraint that ensures the translated code is causally correct in the Lustre sense.

Question 13 Can one of the constructs (weak *versus* strong) be expressed in terms of the other?

Question 14 [*] Extend the reaction semantics for this new construct. Prove that your translation preserves the semantics.

Exceptions

The kernel language is now extended with a programming construct to raise and trap exceptions.

$$D \quad ::= \quad \texttt{exit}\ T \mid \texttt{try}\ D\ \texttt{with}\ \texttt{|}\ T\ \texttt{then}\ D\ \ldots\ \texttt{|}\ T\ \texttt{then}\ D\ \texttt{done}$$

`exit` T raises the exception with name T (we suppose that exception names are declared globally). The construct `try` D `with |` T_1 `then` D_1 ... `|` T_n `then` D_n `done`, where $T_1, \ldots, T_n$ are supposed to be pairwise distinct, executes D and at the instant T_i is raised, the corresponding block D_i becomes active for the rest of the execution.

An exception T can be raised several times in a single instant (e.g., `exit` T `and exit` T) with the same effect as a single raise. Two different exceptions can also be raised simultaneously (e.g., `exit` T_2 `and exit` T_1): the first matching handler in the list of handlers is activated (here D_1).

Exceptions are an essential feature to deal with partially defined functions, e.g.:

```
node safe_div(x, y) returns o
  if y = 0 then exit Div_by_zero else o = x / y done
```

The specification is voluntarily left informal so that you have full freedom to interpret the exception mechanism on you own, to propose a semantics, a translation and/or a compilation mechanism.

Question 15 What behavior would you propose for the following two programs, in term of input/output values?

```
  if y = 0 then exit Div_by_zero else o = x / y done
and
  k = o + 1
and m = 0 -> pre m + 1
```

and:

```
  trap
    if y = 0 then exit Div_by_zero else o = x / y done
   and
    k = o + 1
   and m = 0 -> pre m + 1
  with
   Div_by_zero -> o = 42 done
  and
   po = 0 -> pre(o)
```

The reaction semantics for equations can be extended by adding the set of exceptions that are raised.

$$R \vdash D \stackrel{R' \mid S}{\longrightarrow} D' \text{ with } R' \subseteq R$$

where $S = \{T_1, ..., T_n\}$ is a set of exception names. Its intuitive meaning is that the set of equations D defines the reaction environment R and raises the set of exceptions S. When no exception is raised, this set is empty.

Question 16 What would you suggest for the reaction semantics for the construction `trap/with` and `exit T`? Illustrate you choice on the three previous examples.

Propose a formal definition for the predicate $R \vdash D \stackrel{R' \mid S}{\longrightarrow} D'$ for the two programming constructs.

Question 17 Propose an encoding of `exit` T and the `trap/with` construct by mean of the programming constructs considered in the previous sections. Extend $Tr(.)$ accordingly.