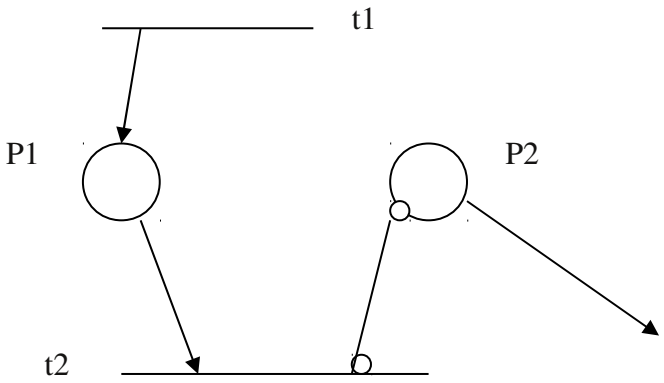
Analisi e progetto di sistemi critici

# Formal Methods

Final test, June 9, 2003

## Exercise 1

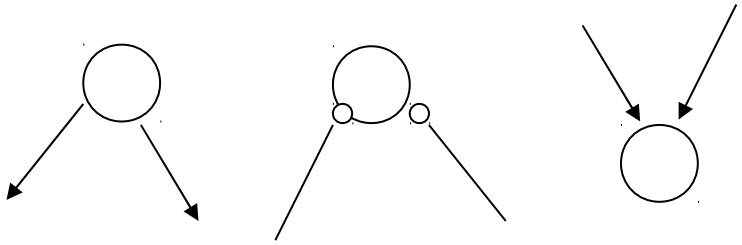
The following fragment of Petri net introduces *inhibitor arcs*. Intuitively, an inhibitor arc is such that the transition connected through it cannot fire if its place contains one or more token.




Thus, transition t2 is enabled if and only if there is at least one token in P1 and no token in P2.

By following the same approach used for timed Petri nets without inhibitor arcs, formalize timed Petri nets with inhibitor arcs through suitable TRIO axioms. You may assume the same simplifying assumtpions used for traditional TPNs, i.e.:

1. 1-bounded PNs
2. No conflicts, i.e., no more than one regular arc and no more than one inhibitor arc outcoming from the same place; no more than one regular arc incoming into any place. (Fragments such as those below are excluded; but the fragment of previous figure is allowed)



Plus the following one.

3. Lowerbounds greater than 0 and upperbounds less than infinity.

**Warning**

The informal timing semantics of inhibitor arcs is subject to the following ambiguous interpretation.

One possible semantics, say SEM1 states that a transition is enabled at the moment when a token exists in “positive places” and no token exists in places with inhibitor arcs. Starting from that time the firing can or must occur within the given time bounds.

Another possible semantics, say SEM2, starts counting time since the time positive places have tokens therein –without taking into consideration places with inhibitor arcs- and the firing can occur only if, at the time of occurrence, the inhibited place has no token, independently on when it has been emptied.

For instance, with reference to the above fragment, suppose that:

transition t2 has time bounds [3,6]

a token is produced in P1 at time 1

a token is in P2 at time 0 and is consumed at time 8

According to SEM1 t2 could fire at any time between 11 and 14

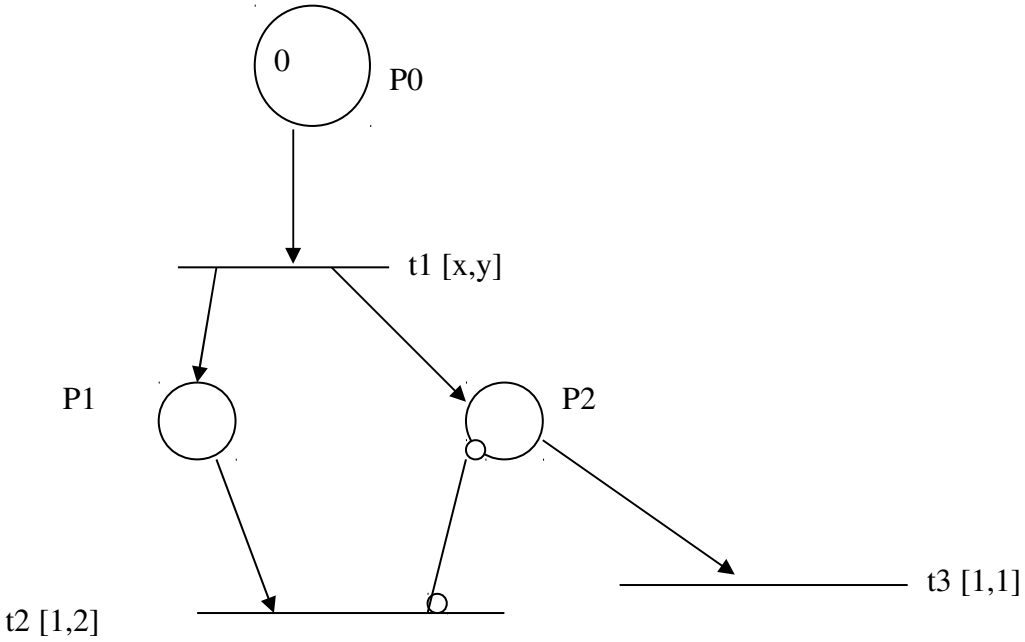
According to SEM2 t2 could never fire because during the interval 4-7 it would be inhibited by the token in P2. If instead the token in P2 were consumed before time 7, t2 would necessarily fire within time 7.

You may choose to formalize either one of SEM1 or SEM2.

As an **option** you may briefly discuss which semantics seems more suitable for practical cases. You may also decide to formalize both of them.

## Exercise 2

By using the axioms produced for exercise 1 –and the other axioms already introduced for TPNs- prove that in the fragment below, if transition t1 fires at a given instant, then transition t2 fires within 3 time units after t1's firing . Notice that such a property is true under both semantics SEM1 and SEM2.



**Solution of Exercise 1.(according to SEM2)**

As usual, let $m_{tr}$, $M_{tr}$ denote the lower and upper tim bounds of a generic trnsition tr.

We define the following axioms

(1): fire(r) $\rightarrow$ UpToNow($\neg$fire(r)) (for every transition)

(Mar): marked(P) $\leftrightarrow$ $Since_{ie}$($\neg$ fire(t2), fire(t1)), (for every triple <t1, P, t2> t1 input of P, t2 output of P. Notice that such an axiom holds even if an inhibitor arc is connected to P)

(LB.(t2)) fire(t2) $\rightarrow$ $Lasted_{ie}$ (marked(P1) $m_v$) $\wedge$ $\neg$ marked(P2),

(UB.(t2)) $Lasted_{ie}$ (marked(P1), $M_v$) $\wedge$ $\neg$ marked(P2)$\rightarrow$ fire(t2)

**Solution of Exercise 2.(according to SEM2)**

Assume by contradiction Lasts ($\neg$ fire(t2), k) $\wedge$ k >= 3.Thus,

fire (t1). $\wedge$ Lasts ($\neg$ fire(t2), k) → Lasts (marked (P1), k) (by axiom Mar).→ Lasts (marked (P1), 2)

Let us now prove the lemma: Futr ($\neg$ marked (P2), 2).

- Again by contradiction Futr (marked (P2), 2).→ Futr($Since_{ie}$($\neg$ fire(t3), fire(t1)), 2), but, since t1 fires at the current instant, this would imply $Lasts_{ie}$ ($\neg$ fire(t3), 2) → $Lasts_{ii}$ ($\neg$ fire(t3), 1) and thus $Lasts_{ii}$ (marked (P2), 1), which in turn implies Futr(fire(t3), 1) –through a simple translation of axiom

  (UB.(a)) $Lasted_{ii}$ (marked(P), $M_v$) $\rightarrow$ fire(v) - a contradiction

Now, by –a simple translation of- UB(t2) we obtain Futr(fire(t2),2), a contradiction.

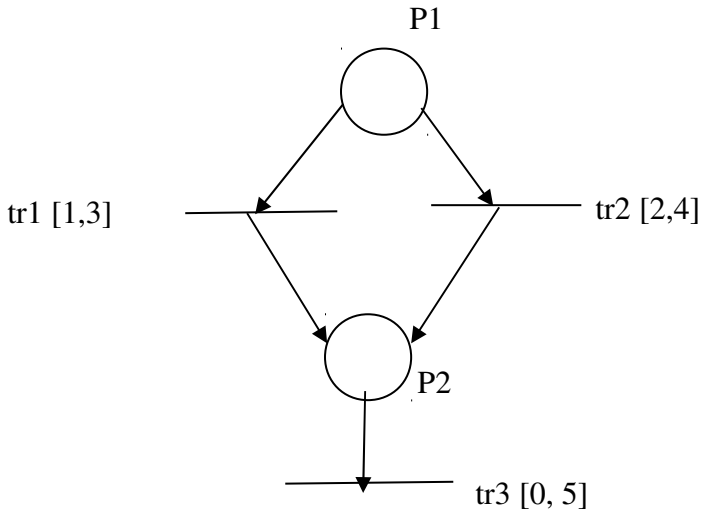
Analisi e progetto di sistemi critici

# Formal Methods

Final, optional test, June 14, 2004

**Exercise**

Consider the following fragment of timed Petri net.




The axiomatization given during the course does not allow to deal with such a net. In particular, there is no way to prove formally that, if at a given time t0 P1 becomes marked, then, tr3 will fire within the time interval [x?, y?] after t0.

Briefly sketch how the axiom system could be extended in such a way as to cope with nets of the above type and how it could be used to prove the related property also stated above. You do not need to go into all details: a clear and succint exposition is enough. However, possible restrictions on the net type should be clearly stated.

*Analisi e progetto di sistemi critici –*

## Formal Methods (UIC)

Mid-term test May 16th, 2005

**Test score**: **12/10**

**NB**: the final score – expressed in 30ths – for the course will be obtained by adding to this score the points you will get from the homework (expressed in 20ths) and, possibly the score of the final – optional – test. The result will be registered as the Italian course and converted into American grades for UIC.

Let x and y be two strings of length < 30. They are stored into two arrays with the same name of 30 characters each. x and y are terminated by the special character '#'.

**Exercise 1 (9/10)**

Specify, write, and prove the partial correctness (plus, give a few hints about termination) of a program that produces a third string z (stored into a suitable array with the same name and terminated by '#' as well) that is the longest prefix that x and y have in common.

**Exercise 2 (3/10; to be done only after completion of exercise 1)**

Briefly sketch, without developing all details, how the previous exercise could be modified so that z is the concatenation of the longest prefix and the longest suffix that x and y have in common.

For instance: x = abacccbaa, y = abcbaccbaa → z = abccbaa

(NB. The special character '#' is not *part* of the string: it is only used as a terminator)

**Sketches of possible solutions (not the only ones!)**

**Exercise 1**

Let's assume that x and y are arrays of 30 characters, with index ranging within [1..30] and z is an array of 30 characters, with index ranging within [1..30].

Suitable pre- and post-conditions can be defined as follows

PRE: $\{(\exists\, h\ (1 \le h \le 30) \wedge x[h] = \text{'\#'} \wedge (\forall j\ (j < h \rightarrow x[j] \neq \text{'\#'}))) \wedge$

$(\exists\, k\ (1 \le k \le 30) \wedge y[k] = \text{'\#'} \wedge (\forall j\ (j < k \rightarrow y[j] \neq \text{'\#'})))\}$

POST: $\{\exists\, h, k, m\ ($

$(1 \le h \le 30) \wedge x[h] = \text{'\#'} \wedge (\forall j\ (j < h \rightarrow x[j] \neq \text{'\#'})) \wedge$

$(1 \le k \le 30) \wedge y[k] = \text{'\#'} \wedge (\forall j\ (j < k \rightarrow y[j] \neq \text{'\#'})) \wedge$

$(1 \le m \le 30) \wedge z[m] = \text{'\#'} \wedge (\forall j\ (1 \le j < m \rightarrow z[j] \neq \text{'\#'}))$

$\wedge$

$(\exists\, p\ (1 \le p \le 30) \wedge$ /* p is the length of the common prefix of x and y + 1*/

$(\forall j\ ((1 \le j < p \rightarrow (x[j] = y[j] = z[j])) \wedge (x[p] \neq y[p] \vee p = h = k = m))$

A Pascal-like program that computes z (assuming that x and y are already stored in memory) is the following:

```
begin
i := 1;
while x[i] = y[i] and x[i] ≠ '#' do z[i] := x[i]; i := i+1 od
z[i]:= '#'
end
```

A fairly natural invariant for a partial corrrectnes proof is the following, whose kernel states that z always stores a common prefix of x and y:

I: $(\forall j\ ((1 \le j < i \rightarrow (x[j] = y[j] = z[j]))) \wedge \ldots$

In order to carry over the proof it should be completed with other –obviously invariant- assertions that are omitted here for simplicity (e.g. the part of the postcondition that defines p as the length of the common prefix of x and y + 1).

The proof that I is indeed invariant follows a standard path; as well as the fact that it is obviously implied by initialization.

At loop exit either x[i] ≠ y[i] which easily leads to i = p = m; or x[i] = ‘#’ which implies i = h and therefore also i = m; which provides the full postcondition after the last statement z[i]:= ‘#’.

**Exercise 2**

Notice that now z is an array of 60 characters.

The precondition is the same as in Exercise 1. The postcondition can be modified as follows.

POST: {$\exists$ h, k, m

$((1 \le h \le 30) \wedge x[h] = \text{‘\#’} \wedge (\forall j\ (j < h \rightarrow x[j] \neq \text{‘\#’}))) \wedge$

$(1 \le k \le 30) \wedge y[k] = \text{‘\#’} \wedge (\forall j\ (j < k \rightarrow y[j] \neq \text{‘\#’}))) \wedge$

$(1 \le m \le 60) \wedge z[m] = \text{‘\#’} \wedge (\forall j\ (1 \le j < m \rightarrow z[j] \neq \text{‘\#’})))$

$\wedge$

($\exists$ i, p $(1 \le i, p \le 30) \wedge$ /* i and p mark the common prefixes and suffixes of x and y, resp.*/

/*the prefix of z is the common prefix of x and y. The suffix of z is the common suffix of x and y. Furthermore z does not contain any other character besides those, i.e., its length is the sum of the two lengths. Notice that, if common prefix and suffix have a common substring, it is repeated.*/

$\alpha$) $(\forall j\ ((1 \le j < i \rightarrow (x[j] = y[j] = z[j])) \wedge (x[i] \neq y[i] \vee x[i] = y[i] = \text{‘\#’}$ /* i = h = k*/))

$\wedge$

$\beta$) $(\forall j\ ((1 \le j < p) \rightarrow (x[h-j] = y[k-j] = z[m-j])) \wedge (p < 30 \wedge x[h-p] \neq y[k-p] \vee p = h = k)$

$\wedge\ m = i + p - 1$

The program, and, consequently, the proof can be split into the following fragments:

1. Compute i and p –and prove the corresponding assertion
2. Write z up to i – 1 by copying x --- and prove assertion $\alpha$)
3. Write z from i to i + p - 1 by copying y’s elements in reverse order --- and prove assertion $\beta$)
4. Write ‘#’ --- and prove m = i + p -1

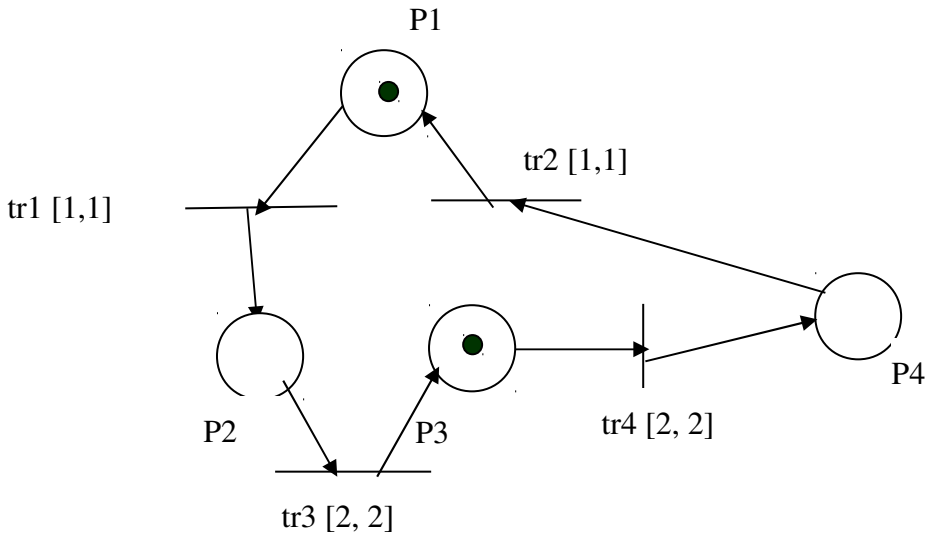
# Analisi e progetto di sistemi critici

# Formal Methods (Master UIC)

Final, optional test, June 15, 2005

**Exercise**

Consider the following timed Petri net.




Assume that the net is initialized at time t0 with the marking of the figure.

1. Express in TRIO the property that the behavior of the net is such that no place will ever contain more than one token.
2. Briefly sketch a formal proof of the above property, **if it holds**.

Notice that, in order to carry over your job, you may need to slightly modify the axiom system that was provided in the class for highly simplified timed Petri nets. In such a case, you do not have to provide a full axiomatization suitable to cope with any general net and any possible property: it is sufficient that you provide enough definitions, axioms, and rules so that the above property can be proved.

You do not need to go into all details: a clear and succint exposition is enough. However, possible restrictions on the net type your axiomatization is able to handle should be clearly stated.

**Hint for a possible solution**

The simplified axiomatization assumes a priori that the net is 1-bounded. Here, in principle, we must also formalize the fact that more than one token is stored in a place: this just to *express* the fact that such a situation does not occur.

If we do not strive for full generality, we can formalize this fact by a formula that states that a place is "doubly-marked" iff sometimes in the past –including the current instant- it was singly-marked and its input transition fired and its output transition did not fire since then, –again, including the current instant-.

With such a new definition, one can easily express the desired property.

The axiom system should be slightly extended to formalize the behavior with multiply marked places. However, since in this case we are only interested in proving that our net will never be doubly marked in any place, it is sufficient to use axioms specifying the behavior of singly marked places and then showing that conditions leading to doubly marking never occur.

At this point the proof of the desired property can be carried over along several guidelines. For instance:

1. Proving a –fairly obvious- periodic behavior
2. Proving that during a whole period no place ever get doubly marked. This in turn can be proved, e.g., by showing that the necessary condition that lead to doubly marking would imply some contradiction with te stated hypothesis on initial marking. Such a proff is fostered by the deterministic behavior of this particular net.

*Analisi e progetto di sistemi critici –*

## Formal Methods (UIC)

Mid-term test May 10th, 2006

**Test score**: **13/10ths**

**NB**: the final score – expressed in 30ths – for the course will be obtained by adding to this score the points you will get from the homework (probably expressed in 12ths) and the score of the final test (probably expressed in 8ths). The result will be registered as the Italian course and converted into American grades for UIC.

**Exercise**

Assume that two arrays *a* and *b*, both of *n* elements – say, integers –, are stored in memory.

Give a formal specification in terms of suitable pre- and post-conditions for a program –fragment- that:

1) checks whether *a* or *b* have repeated occurrences of the same element and stores the result of such a check in a boolean flag *F1* (true if neither *a* nor *b* have repeated elements);
2) if neither *a* nor *b* have repeated elements, checks whether they are a permutation of each other and stores the result in another boolean flag *F2*.

Write a program fragment – in a suitable "toy subset" of a normal programming language such as Pascal, C, etc.- that satisfies the above specification.

Prove the partial correctness of the program you wrote.

**Suggestion (absolutely not mandatory!)**

You may split your program –and your specification, accordingly- into two components devoted to requirement 1) and 2), respectively. You can then structure your proof in the same way. In case you feel running short of time we recommend that you first set up the "structure of the exercise", i.e. pre- and post-conditions, program, assertions in the critical points of the program, invariants; then, you may work out the details of the proof by giving priority to the second part of the proof; only later, if more time is left, you may also detail the first part of the proof.

**NB**. You can get a good score even if you do not fill up all details!

**Sketch of a possible solution (not the only one!)**

```
{ n > 0 }
begin
      F1 := true;  i := 0;
      while i < n do
            j := i+1;
            while j < n do
                  if a[i] = a[j] or b[i] = b[j] then
                        F1 := false; fi;
                  j := j+1; od;
            i := i + 1; od;
      n_eq := 0; i := 0; F2: = true;
      if F1 then
            while i < n and F2 do
                  j := 0; F3 := false;
                  while j < n and not F3 do
                        if b[j] = a[i] then
                              F3 := true;
                        else j := j+1; fi; od;
                  F2 := F3; i := i+1; od;
      fi; F2 := (i = n) and F2;
end;
```

$\{ (F1 \leftrightarrow \forall x,y\ (0 \le x < y < n \rightarrow a[x] \neq a[y] \wedge b[x] \neq b[y])) \wedge$

$F2 \leftrightarrow F1 \wedge \forall x\ (0 \le x < n \rightarrow \exists\ y\ (0 \le y < n \wedge a[x] = b[y]))\ \}$

## Loop Invariants:

Loop invariant for the first outer **while**:

$I_1 = \{ (F1 \leftrightarrow \forall x,y\ (0 \le x < i \wedge 0 \le y < n \wedge x \neq y \rightarrow a[x] \neq a[y] \wedge b[x] \neq b[y])) \wedge i \le n \}$

Loop invariant for the first inner **while**:

$J_1 = \{ I_1 \wedge (F1 \leftrightarrow \forall y\ (i < y < j \rightarrow a[i] \neq a[y] \wedge b[i] \neq b[y])) \wedge i < j \le n \}$

After the first (outer) while, we are able to prove:

$K = \{ (F1 \leftrightarrow \forall x,y\ (0 \le x < y < n \rightarrow a[x] \ne a[y] \wedge b[x] \ne b[y])) \}$

Loop invariant for the second outer **while**:

$I_2 = \{ F2 \leftrightarrow \forall x\ (0 \le x < i \rightarrow \exists y\ (0 \le y < n \wedge a[x] = b[y])) \wedge i \le n \}$

Loop invariant for the second inner **while**:

$J_2 = \{ I_2 \wedge j \le n \wedge (F3 \rightarrow \exists y\ (0 \le y < n \wedge a[i] = b[y])) \wedge$

$(\neg F3 \rightarrow \forall y\ (0 \le y < j \rightarrow a[i] \ne b[y])) \}$

## Partial Correctness Proof:

We split the proof into:

1. { Pre } `F1:=true; i:=0;` { $I_1$ }
2. { $I_1 \wedge i < n$ } *first outer loop body* { $I_1$}
3. { $I_1 \wedge i \ge n$ } $\rightarrow$ {K}
4. {K} `n_eq:=0; i:=0;` {$(I_2 \vee \neg F1) \wedge K$}
5. {$(I_2 \vee \neg F1) \wedge K$} **`if`** … **`fi;`** {$\neg F1 \vee (I_2 \wedge i \ge n)$}, which is reduced (after trivial case discussion) to: { $I_2 \wedge i < n \wedge F2$ } *second outer loop body* { $I_2$ }
6. {$(\neg F1 \vee (I_2 \wedge (i \ge n \vee \neg F2))) \wedge K$} `F2:= (i = n)` **`and`** `F2;` {Post}

**Steps 1, 3, 4, and 6** are trivial.

Let us consider **step 2**, which is:

**2.** { $I_1 \wedge i < n$ } `j:=i+1;` **`while`** … **`od;`** `i:=i+1;` {$I_1$}

After backward substitution through the last statement, $I_1$ becomes:

$I_1' = \{ (F1 \leftrightarrow \forall x,y\ (0 \le x \le i \wedge 0 \le y < n \wedge x \ne y \rightarrow a[x] \ne a[y] \wedge b[x] \ne b[y])) \wedge i < n \}$

So, we prove:

**2.1.** { $I_1 \wedge i < n$ } `j:=i+1;` { $i < n \wedge J_1$ }

**2.2.** { $i < n \wedge J_1 \wedge j < n$ } **`if`** … **`fi;`** `j:=j+1;` { $I_1 \wedge i < n \wedge J_1$ }

**2.3.** { $I_1 \wedge i < n \wedge J_1 \wedge j \ge n$ } $\rightarrow$ { $I_1'$ }

**2.1.** is easy, once we notice that the interval $i < y < i+1$ is empty.

**2.2.** is split into then and else branches, that is, respectively:

**2.2.1.** { $i < n \wedge J_1 \wedge j < n \wedge (a[i]=a[j] \vee b[i] = b[j])$ }

`F1 := false; j:=j+1;` { $I_1 \wedge i < n \wedge J_1$ }

**2.2.2.** { $i < n \wedge J_1 \wedge j < n \wedge (a[i] \neq a[j] \wedge b[i] \neq b[j])$ } `j:=j+1;` { $I_1 \wedge i < n \wedge J_1$ }

**2.2.2.** is done straightforwardly by backward substitution; for **2.2.1.** note that $(a[i]=a[j] \vee b[i] = b[j]) \wedge i < j$ implies $\exists y\ (i < y \leq j \wedge (a[y] = a[i] \vee b[y] = b[i]))$, for y = j, as required by the backwardsubstituted expression, as F1 is false in this branch.

For **2.3**, note that we have j = n, so $I_1$ and $J_1$ imply $I_1$' (in particular, note that $I_1$ implies that $\forall y\ (0 \leq y < i \rightarrow a[i] \neq a[y] \wedge b[i] \neq b[y]$, by "switching" x and y).

Let us finally consider **step 5**, which is:

**5.** { $I_2 \wedge i < n \wedge F2$ } `j:=0; F3:=false; while … od; i:=i+1;` { $I_2$ }

This is split into:

**5.1.** { $I_2 \wedge i < n \wedge F2$ } `j:=0; F3:=false;` { $J_2 \wedge i < n$ }

**5.2.** { $J_2 \wedge i < n \wedge j < n \wedge \neg F3$ } **`if ...fi;`** { $J_2 \wedge i < n$ }

**5.3.** { $J_2 \wedge i < n \wedge (j \geq n \vee F3)$ } `F2:=F3; i:=i+1;` { $I_2$ }

Let us consider **5.3.**: by backward substituting $I_2$ we get:

{ $I_2$' } = { $F3 \leftrightarrow \forall x\ (0 \leq x \leq i \rightarrow \exists y\ (0 \leq y < n \wedge a[x] = b[y])) \wedge i < n$ }

We have to show { $J_2 \wedge i < n \wedge (j \geq n \vee F3)$ } $\rightarrow$ { $I_2$' }.

If F3 is false, then $j \geq n$, so j = n, and we have to show that $\exists x\ (0 \leq x \leq i \wedge \forall y\ (0 \leq y < n \rightarrow a[x] \neq b[y]))$. Notice that F3 false and $J_2$ imply $\forall y\ (0 \leq y < j=n \rightarrow a[i] \neq b[y])$, so the goal follows for x = i.

If F3 is true, then we have to show that $\forall x\ (0 \leq x \leq i \rightarrow \exists y\ (0 \leq y < n \wedge a[x] = b[y]))$. From $J_2$, we have $\exists y\ (0 \leq y < n \wedge a[i] = b[y]))$ and from $I_2$ (which is part of $J_2$), we have $\forall x\ (0 \leq x \leq i \rightarrow \exists y\ (0 \leq y < n \wedge a[x] = b[y]))$: The goal follows by combining the two formulas.

Let us move to **5.1.**, which amounts to proving that $\{ I_2 \wedge i < n \wedge F2 \}$ implies $J_2$ where we substitute 0 for j and **false** for F3, that is:

$$\{ I_2 \wedge j \leq n \wedge \forall y\ (0 \leq y < 0 \rightarrow a[i] \neq b[y]) \} \equiv \{ I_2 \wedge j \leq n \}$$

So the implication is immediate (as the interval $0 \leq x < 0$ is empty).

Let us finally consider the longer step **5.2.**; after using the inference rule for the if, it becomes:

**5.2.1.** $\{ J_2 \wedge i < n \wedge j < n \wedge b[j] = a[i] \wedge \neg F3 \}$ `F3:=true;` $\{ J_2 \wedge i < n \}$

**5.2.2.** $\{ J_2 \wedge i < n \wedge j < n \wedge b[j] \neq a[i] \wedge \neg F3 \}$ `j:=j+1;` $\{ J_2 \wedge i < n \}$

For **5.2.1.**, let us backward substitute through the assignement, thus reducing to prove that $\{ J_2 \wedge i < n \wedge j < n \wedge b[j] = a[i] \wedge \neg F3 \}$ implies:

$$\{ I_2 \wedge j \leq n \wedge \exists y\ (0 \leq y < n \wedge a[i] = b[y])) \}$$

$I_2$ is part of $J_2$, and $j < n$ is stronger thhan $j \leq n$. Moreover, $\exists y\ (0 \leq y < n \wedge a[i] = b[y]))$ is true for $y = j < n$, since $b[j] = a[i]$.

For **5.2.2.**, let us backward substitute through the assignement, thus reducing to prove that $\{ J_2 \wedge i < n \wedge j < n \wedge b[j] \neq a[i] \wedge \neg F3 \}$ implies:

$$\{ I_2 \wedge j < n \wedge (F3 \rightarrow \exists y\ (0 \leq y < n \wedge a[i] = b[y])) \wedge (\neg F3 \rightarrow \forall y\ (0 \leq y \leq j \rightarrow a[i] \neq b[y])) \}$$

Note that F3 is false, so basically we just have to prove $\forall y\ (0 \leq y \leq j \rightarrow a[i] \neq b[y])$, as $I_2$ and $j < n$ are both in the antecedent and the consequent.

$J_2$ and $\neg F3$ imply that $\forall y\ (0 \leq y < j \rightarrow a[i] \neq b[y])$; moreover we have $b[j] \neq a[i]$. So, overall, $\forall y\ (0 \leq y \leq j \rightarrow a[i] \neq b[y])$, as required.

## Comments:

Another, faster, solution would have ordered the two arrays, so that both checks can be performed in linear time. The partial correctness proof could have reused the results from the ordering algorithms.

# Analisi e progetto di sistemi critici

# Formal Methods (UIC)

Final test, June 14, 2006

**Exercise 1 (points 5/8ths)**

Consider the following version of timed PNs:

A constant time is attached to each place. Its meaning is that, once a token has been produced in place P with associated time k, it can be used to fire an output transition of P only after k time units elapsed since its creation.

Formalize the behavior of this type of nets through suitable TRIO axioms. You may assume reasonable simplifying assumptions, if needed (e.g. the same as we adopted to formalize PNs à la Merlin and Farber).

**Exercise 2 (points 6/8ths; to be done only *after Exercise 1!*)**

With reference to Exercise 1, assume now that a *pair* of constant values is attached to each place P: $[m_P, M_P]$, whose intuitive meaning is that a token must abide in P at least $m_P$ but not longer than $M_P$, ***if the*** (***an***, in the general case) ***output transition of P can fire within such an interval***.

Axiomatize this enriched version of TPN with time associated with places. In doing so you may notice that a few ambiguities may arise when interpreting the above intuitive semantics. In particular, what happens if a token has not been consumed within the time associated with the place where it is? Does it stay there forever or is it simply "destroyed"? If it stays there is it possible to deposite another token in the same place still assuming the net as 1-bounded (given that the previous token is definitely lost)? Clearly point out such possible ambiguities; take a choice you deem as reasonable –with appropriate short explanation- and formalize the net behavior accordingly.

**Solution of Exercise 1**

Let us adopt the same simplifying assumptions as for M&F TPNs. This means that we should consider only fragments such as those depicted below:

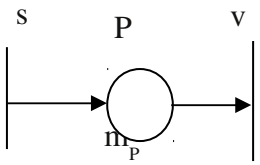



Figure (a)

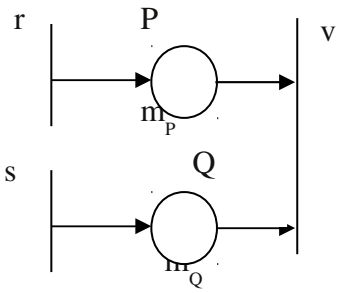



Figure (b)

Furthermore each place can store at most one token.

Under these assumptions, a suitable set of axioms to formalize the behavior of such nets is the following:

(Ist): $\text{fire}(r) \rightarrow \text{UpToNow}(\neg \text{fire}(r))$

(Mar): $\text{marked}(P) \leftrightarrow (\text{Since}_{ie}(\neg\ \text{fire}(v), \text{fire}(s)) \vee (\text{fire}(s) \wedge \neg\ \text{fire}(v))$

(LB.(a)) $\text{fire}(v) \rightarrow \text{Lasted}_{ie}\ (\text{marked}(P), m_P)$

(LB.(b)) $\text{fire}(v) \rightarrow \text{Lasted}_{ie}\ (\text{marked}(P), m_P) \wedge \text{Lasted}_{ie}\ (\text{marked}(Q), m_Q)$

Notice that the absence of upperbounds *de facto* produces a weak time semantics with little difference w.r.t. untimed PNs (only delays in the firing sequences are introduced).

**Solution of Exercise 2**

First, we must decide what happens with "lost tokens", if any. A reasonable assumption is that they are destroyed if not consumed within the given interval: in such a way they do not hamper the behavior of the net, still allowing it to remain 1-bounded. A different choice would be less "dramatic" if the constraint of 1-boundedness were relaxed.

Next, notice that the behavior of the net fragment depicted in Figure (a) now becomes the same as in the M&F nets: it is sufficient to add the axiom:

(UB.(a)) $\text{Lasted}_{ie}(\text{marked}(P), M_P) \rightarrow \text{fire}(v)$

Also, axiom Mar is still valid for this fragment.

The situation becomes a little more intricate for the fragment of Figure (b). If we assume that, once a token cannot be used to fire its output transition anymore, it "disappears", the axiom Mar should be rewritten as:

(Mar): $(\text{Up\_to\_now}\ (\neg\ \text{marked}(P) \wedge \text{fire}(r) \wedge \text{Lasts}_{ie}(\neg\ \text{fire}(v), k) \wedge k \leq M_P) \rightarrow$

$\text{Lasts}_{ie}(\text{marked}(P), k)$

$\wedge$

$(\text{Up\_to\_now}\ (\text{marked}(P) \wedge \text{fire}(v)) \rightarrow \text{Now\_On}\ (\neg\ \text{marked}(P))$

$\wedge$

$\text{Lasted}_{ie}(\text{marked}(P), M_P) \rightarrow \text{Now\_On}\ (\neg\ \text{marked}(P))$

$\wedge$

$\neg\ \text{marked}(P) \wedge \text{Lasts}_{ie}(\neg\ \text{fire}(r), k) \rightarrow \text{Lasts}_{ie}\ (\neg\ \text{marked}(P), k)$

With a similar axiom for place Q and transition s.

Finally the upperbound axiom for fragment (b) can be restated in the following way:

(UB.(b)) $\text{Lasted}_{ie}(\text{marked}(P), M_P) \wedge (\text{marked}(Q))$

$\vee$

$\text{Lasted}_{ie}(\text{marked}(Q), M_Q) \wedge (\text{marked}(P))$

$\rightarrow \text{fire}(v)$

Notice that a different –still strong time- semantics could impose a transition to fire only when both input places have been marked for their maximum time interval. In this case, however, we should allow a token to stay in its place for more than the maximum time and still be usable if it could not be consumed earlier.

*Analisi e progetto di sistemi critici –*

## Formal Methods (UIC)

## Mid-Term Test: Hoare's method part

May 9th, 2007

*The present mid-term test accounts for about 1/3 of the whole exam. Thus, its evaluation will be given in 10ths. However, particularly good solutions could obtain even more than 10/10ths.*

Consider the following problem:

Given an array of $n$ positive integers, determine whether or not there are two elements $x, y$ in the array such as $x = y^2$.

Formally specify the requirements for an algorithm to solve such a problem, in terms of pre- and post-condition; then code an algorithm by using a suitable pseudocode (e.g. Pascal- or C-like) and prove its partial correctness.

**Remarks**

1. Any reasonable algorithmic solution –with related proof– will be accepted; however, one that optimizes efficiency in terms of asymptotic time complexity will provide extra benefits in the evaluation.
2. There is no need to develop all details of the proof –and of the algorithm's code: in particular the use of suitable "library" algorithms, provided they are well and formally specified, does not require their coding and correctness proof.

**Hints for possible solutions**

The Hoare's style specification in terms of Pre- and Post- conditions can be formalized as follows:

$\{n >= 1 \wedge \forall i(1 <= i <= n \rightarrow a[i] > 0)\}$

P

$\{found \leftrightarrow \exists j,k(1 <= j <= n \wedge 1 <= k <= n \wedge j \neq k \wedge a[j] = a[k]^2\}$

We outline two possible solutions, with different efficiency.

1. A standard solution would be based on two nested loops that scan the whole array:

- `found` is initialized to false
- for every `j`, `a` is scanned with index `k` to see whether $a[k]^2 = a[j]$ for some `k` (and `k` is different than `j`)
- if and when such a `k` is found, variable `found` is set to true and loops are exited (obviously they are exited at the end of the array if the equality test is failed).

This solution has an asymptotic worst-case time complexity of $\Theta(n^2)$.

**Correctness proof sketch**

The *core* of a suitable invariant for the *external* loop (additional details depend on implementation details), assuming that `j` is the running index of the external loop and `k` for the internal one, could be:

$(\neg found \rightarrow \forall w(1 <= w < j \rightarrow \neg\exists z(1 <= z <= n \wedge z \neq w \wedge a[w] = a[z]^2)))$

$\wedge$

$(found \rightarrow \exists z(1 <= z <= n \wedge a[j] = a[z]^2 \wedge z \neq j))$

Note that we assume that the loop is exited *immediately after* found is set to true, otherwise the second conjunct of the invariant should include an existential quantification over the range `[1..j)`.

Similarly, the *core* of a suitable invariant for the *internal* loop could be:

$I = \{ (\neg found \rightarrow \forall w(1 <= w < j \rightarrow$

$(\neg\exists z(1 <= z <= n \wedge z \neq w \wedge a[w] = a[z]^2))$

$\wedge (\neg\exists p(1 <= p < k \wedge p \neq j \wedge a[j] = a[p]^2))))$

$\wedge$

$(found \rightarrow k \neq j \wedge a[j] = a[k]^2) \}$

Also for the internal loop we assume it's exited *immediately after* found is set to true.

2. A more efficient solution would consist in:

- sort the array `a`
  - sorting should be performed by means of $\Theta(n.\log(n))$ algorithm, such as merge-sort
- for every `j`, search the array `a` for an element `a[k]` such that `a[j] = a[k]`$^2$
  - searching should be performed by by applying a suitable modification of a binary search algorithm of logarithmic complexity;
  - since this search should be repeated for every `j` in the worst case, the total complexity of this step is $\Theta(n.\log(n))$

Overall, this solution has an asymptotic worst-case time complexity of $\Theta(n.\log(n))$.

**Correctness proof sketch**

By implicitly assuming that the contents of array `a` is not altered by the sorting algorithm, but only re-ordered, we can split our job into the following steps:

`{n >= 1 ∧ ∀i(1 <= i <= n → a[i] > 0)}`

SORT

`{n >= 1 ∧ ∀i(1 <= i <= n → (a[i] > 0`

`∧ ∀j(1 <= j < n → a[j] <= a[j+1]))) }`

SEARCH

`{found ↔ ∃j,k(1 <= j <= n ∧ 1 <= k <= n ∧ j ≠ k ∧ a[j] = a[k]`$^2$`}`

By skipping the implementation and correctness proof of the SORT part – as it's already provided in suitable "libraries" (see e.g. the text-book) – the SEARCH part could consist, for each `j`, in a suitable adaptation of the classsical binary search algorithm, such as the following one. Moreover, a preliminary check should verify whether one of the particular cases `a[1] = a[2] = 1` or `a[1] = 1 ≠ a[2]` occurs. If `a[1] = a[2] = 1` the search is already finished succesfully; otherwise, the following algorithm is run for each `j` starting from 2 if `a[1] = 1 ≠ a[2]` and starting from 1 otherwise.

```
l:=1; u:=n+1;
while l < u do
      k := (l+u-1) div 2;
      if a[k]^2 < a[j]
            then l := k+1
            else u := k
      fi
od
```

After dealing separately with the special case `a[1] = a[2] = 1`, the correctness proof could be centered on a *core* invariant for the outer loop (the one over `j`) similar to invariant I under the additional hypothesis `¬(a[1] = a[2] = 1)`, and on the other following *core* invariant for the inner loop (i.e., the binary search itself) under the same hypothesis:

$$l <= u \wedge \forall p((1 <= p < l \rightarrow a[p]^2 < a[j]) \wedge (u <= p <= n \rightarrow a[p]^2 >= a[j]))$$
$$\wedge \forall h(1 <= h < n \rightarrow a[h] <= a[h+1])$$

## Analisi e Progetto di Sistemi Critici: seconda parte

## Formal Methods: second test

June 19th, 2007

## Exercise

**This exercise is worth 12 points if you do not use the hint that is available at your request; if you use the hint you can obtain at most 8 point.**

**You are given one hour for this exercise.**

In the –simplified- axiomatization of Merlin & Farber timed PNs we used the following axiom to define the fact that a place is marked (in the case of 1-bounded nets):

(Mar): $\text{marked}(P) \leftrightarrow (\text{Since}_{ie}(\neg\ \text{fire}(v), \text{fire}(s)) \vee (\text{fire}(s) \wedge \neg\ \text{fire}(v))$

where s and v are, respectively, the *only* input and output transitions of P.

We also commented that, through this axiomatization *marked* is a predicate that can hold only in left-closed, right-open intervals.

There is however, a particular case where the above axiomatization would be inadequate (in the sense that it would produce behaviors of the net that do not correspond to the intuitive semantics one would like to attach to the net: precisely it would make the presence of the place practically useless).

Point out such a particular case and suggest how you could fix the above axiomatizaztion in order to avoid the undesired semantics. Also, briefly discuss the consequences of your choice in terms of the property of the marking condition under the new formal semantics.

### Warning

Notice that, in order to focus our aximomation on the essential aspects of timed PNs semantics, we overlooked formalizing the initial marking of the net. You may do the same here. However, if you feel more comfortable with a more precise problem definition you may assume (as we did in the KRC example) that for an initially marked place P

(SomP (marked(P))

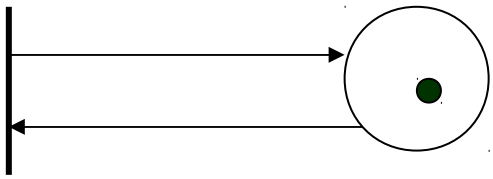
holds and the above axiom (Mar) is augmented as:

(Mar1): $\text{marked}(P) \leftrightarrow (\text{SomP}\ (\text{marked}(P) \wedge \text{AlwP}(\neg\ \text{fire}(v))) \vee$

$(\text{Since}_{ie}(\neg\ \text{fire}(v), \text{fire}(s)) \vee (\text{fire}(s) \wedge \neg\ \text{fire}(v))$

**Hint**

The exception arises if the input and output transitions of P coincide:



**Solution**

In the case of the figure above, by applying the normal axiom we would obtain that P is never marked if not initially marked or it becomes unmarked right after the first firing of v and then remains unmarked forever since both

$Since_{ie}(\neg\ fire(v), fire(s))$ and

$(fire(s) \wedge \neg\ fire(v)$

cannot be satisfied if s = v (notice that $Since_{ie}(\neg\ fire(v), fire(s))$ requires fire(s) in the *past* and then no firing of v since then: impossible if s = v).

Notice also that in any case we do not admit a 0 firing time for s = v since this would be a loop of 0-time transitions.

A possible escape is to assume that a marking holds only in open time intervals. This could be achieved by modifying axiom Mar as:

(Mar2): $marked(P) \leftrightarrow (Since_{ei}(\neg\ fire(v), fire(s))$

(and Mar1 similarly for the initialization). I.e., P is marked if and only if its input transition fired in the past and its output transition did not fire *yet* (including now).

Thus, at the instants when a transition fires, its output place is not marked.

As a further comment, this assumption would be quite reasonable for dense time models; in the case of discrete time models it would have the consequence that if say, s fires at 0 and v at time 1, then P is marked at 1 and not at 0: acceptable but p erhaps not very intuitive.

# Analisi e Progetto di Sistemi Critici

# Formal Methods

July 17th, 2007

**NB: the assigned time is 2 hours and 30 minutes.**

**The score obtained will be added to the one obtained in the homework to produce the final score of the whole exam.**

Some electric car windows behave in the following way:

- There is a button which can be pushed in two different ways: push up and push down;
- The user can push the button –in either direction- either "*instantaneously*" or can he keep it pushed for a while;
    - If the button is pushed instantaneously the window starts moving in the chosen direction and keeps moving until either it reaches the fully closed (or fully open) position or the button is pushed again *in any way*.
    - If the button is kept pushed for a while, the window starts moving in the chosen direction and keeps moving until either it reaches the fully closed (or fully open) position or the button is released.

**Exercise A (9 points)**

Formalize the above rules by using TRIO (the use of TRIO+ or ArchiTRIO to describe the "structure of the system" would be welcome but is by no way mandatory).

For the sake of simplicity you may assume that the speed of the window is the same and constant (say, *k* cm/sec) in both directions.

**Suggestion**

One possible way of formalizing the notion of "*instantaneous push*" is to assume that it lasts less than a –small- time quantity (e.g. 1/10 sec).

Alternatively, you may assume it is an event in the traditional mathematical meaning of the term and formally distinguish between *instantaneous push* and *lasting push*.

In this case the former approach seems more realistic, but the latter one would be welcome as well.

**Exercise B (9 points)**

Give and formalize some sufficient conditions that guarantee that, in whichever position $p$ is the window at any time, it reaches the fully closed position within $h$ seconds (of course, you must state a relation among the speed of the window, the distance between fully closed and fully open positions, and the time $h$).

Then, build a *formal proof* of such a guarantee.

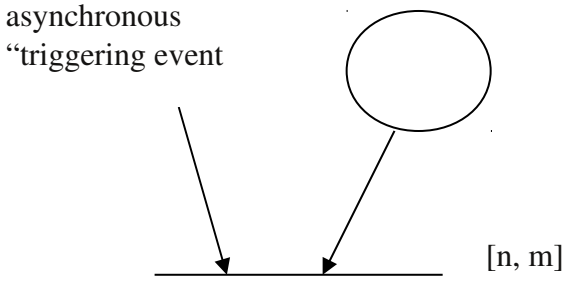
**Exercise C (6 points: it is highly recommended that you do this exercise only if and after you are confident of a good solution of exercises A and B)**

Use a timed Petri net to formalize *–part of-* the rules regulating the movement of the window. You may use the following simplifying assumptions:

- Consider only one direction of the movement –and, consequently, only one mode of pushing the button (e.g., only upwards);
- Formalize only the stopping condition given by the pushing or releasing pushing the button and neglect the condition given by the reaching of the fully closed position.

**Suggestion:**

You are allowed to enrich the normal syntax of Petri nets by introducing "asynchronous events" modeled by arcs that enter a transition without an input place, as depicted in the figure below.




In this case the timed semantics of the transition firing could be defined as: "once a token is in the input place(s) the transition fires if and only if the triggering event occurs within the interval [n , m] (unless disabled in the mean while of course).

**Sketchy solutions**

**Exercise A**

Define predicates P_U (Push Up) and P_D (Push Down); Stop (the window is stopped); M_U (Moving Up), M_D (moving down).

Define also variable P (position); P = 0 meaning window fully open; P = C, meaning window fully closed.

Assume that "instantaneous pushing" means that pushing lasts less than 1/10 sec. Then *some* movement rules are formalized as follows:

(Lasts (P_U, r) $\wedge$ r < 1/10 $\wedge$ $\neg$ (Lasts (P_U, 1/10)$\wedge$ Stop) $\rightarrow$ Until (M_U, (P_U $\vee$ P_D $\vee$ P = C))

(Lasts (P_U, r) $\wedge$ r $\geq$ 1/10 $\wedge$ Stop) $\rightarrow$ Until (M_U, ($\neg$P_U $\vee$ P = C))

(P = p $\wedge$ Lasts (M_U, s) $\wedge$ s $\leq$ (C – p)/k) $\rightarrow$ Futr (P = p + k*s, s)

(P = C $\wedge$ Lasts ($\neg$(M_U $\vee$ M_D), s)) $\rightarrow$ Futr (P = C, s)

…

**Exercise B**

Two sufficient conditions that guarantee Within (P = C, h) (h = C/k) are:

Lasts (M_U, h)

and

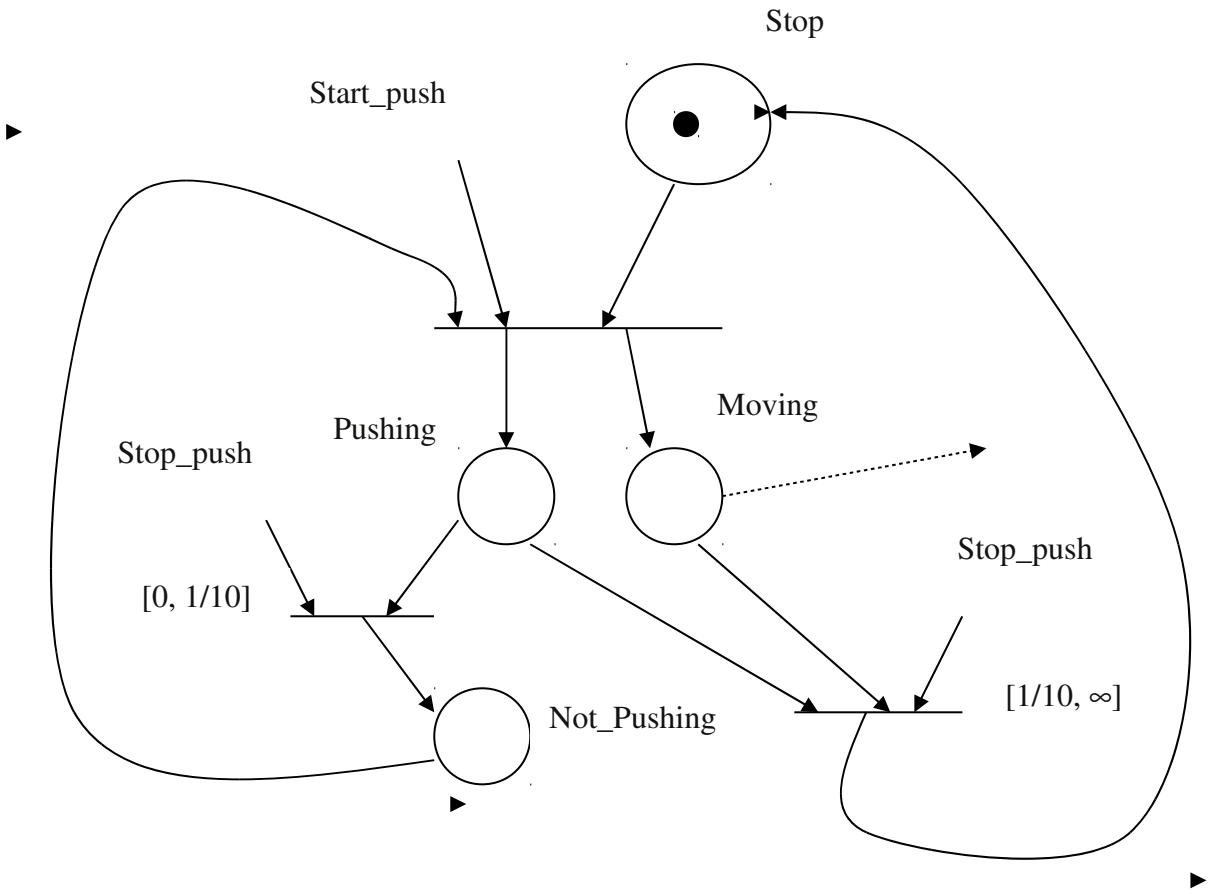
(Lasts (P_U, r) $\wedge$ r $\leq$ 1/10 $\wedge$ Stop $\wedge$ Futr (Lasts (($\neg$P_U $\wedge$ $\neg$P_D), h-r), r)

Of course proving the goal in the first case is easier, though it is not the more comfortable way of achieving it ☺.

**Exercise C**

The following *fragment* uses two events: Start_push and Stop_push (for symmetry, we consider only one direction): it formalizes the facts that if the user starts pushing the button when the window is stopped and stops pushing immediately, then the window keeps moving (until a next pushing); otherwise, the window stops moving as soon as the user stop pushing.

The fragment does *not* formalize the position of the window.


Stop
Start_push
Moving
Pushing
Stop_push
Stop_push
[0, 1/10]
[1/10, ∞]
Not_Pushing

*Analisi e progetto di sistemi critici –*

## Formal Methods (UIC)

## Mid-Term Test: Hoare's method part

May 7th, 2008

*The present mid-term test accounts for about 1/3 of the whole exam. Thus, its evaluation will be given in 10ths. However, particularly good solutions could obtain even more than 10/10ths.*

The following program fragment – coded in C (not in Pascal!) – is intended to copy a string of characters – contained in array `Input` and terminated by the special character `'%'` – into an array `Text` of length `MaxLength`.

- The length of `Input` is greater than `MaxLength` which in turn is greater than 0;
- The string stored in `Input` does not contain further occurrences of `'%'` besides the one terminating the string;
- The whole string must be copied into `Text`, including `'%'`;
- If the input string cannot be fully stored into `Text`, variable `Error` must be set to `TRUE`; otherwise it must be set to `FALSE`.

You are required to:

1. Formally specify the above requirement for the program fragment in terms of Pre and Post conditions.
2. Prove the fragment's correctness with respect to the specification, or disprove it.
3. If the fragment is incorrect with respect to the specification, fix it and prove the fixed version correct.

```
{
  Counter = 0;
  x = Input[0];
  Text[Counter] = x;
  while (x != '%'  &&  Counter < MaxLength)
  {
    Counter++;
    x = Input[Counter];
    Text[Counter] = x;
  }
  if (Counter == MaxLength  &&  x != '%')
    Error = TRUE;
  else Error = FALSE;
}
```

**Sketch of the solution**

A precondition formalizing the hypotheses is the following:

$$\{ \text{MaxLength} > 0 \wedge \exists\, i\ (i >= 0 \wedge \text{Input}[i] = \text{`\%'} \wedge \forall\, j\ (j < i \rightarrow \text{Input}[j] \neq \text{`\%'})\ )\}$$

Then, a postcondition formalizing fragment's requirement is the following:

$$\{\exists\, i\ (i >= 0 \wedge \text{Input}[i] = \text{`\%'} \wedge$$
$$i > \text{MaxLength} - 1 \rightarrow \text{Error} = \text{TRUE} \wedge$$
$$i < \text{MaxLength} \rightarrow (\text{Error} = \text{False} \wedge \forall\, j\ (j \leq i \rightarrow \text{Text}[j] = \text{Input}[j]))$$
$$\}$$

**Notice**: as usual, we implicitly assume that the "input variable `Input`" is not affected by the program fragment.

The proposed fragment is incorrect in a fairly subtle way: in fact, if the input string, including the terminal character `%`, is longer than `MaxLength` it produces an attempt to assign a value to `Text[MaxLength]` which *should* result into a run-time error (but we know that such errors are not always signaled by the execution!), thus making the final check about the value of counter occurring "too late".

To guarantee freedom for such an error one should prove as a precondition to *any* assignment to `Text[Counter]` the invariant assertion $\{0 \leq \text{Counter} \leq \text{MaxLength}-1\}$, which, however, is clearly not guaranteed by the fragment and can be easily disproved (probably in a more apparent way than by testing the program).

Once discovered the error we can simply fix the program in the following (or similar) way:

```
{
  Counter = 0;
  x = Input[0];
  Text[Counter] = x;
  Counter++;
  while (x != '%'  &&  Counter < MaxLength)
  {
    x = Input[Counter];
    Text[Counter] = x;
    Counter++;
  }
  if (Counter == MaxLength  &&  x != '%')
    Error = TRUE;
```

```
  else Error = FALSE;
}
```

The new fragment can be proved correct by means of the following “core invariant” (details are omitted):

{{0 ≤ Counter ≤ MaxLength ∧ ∀ j (j < Counter → Text[j] = Input [j])}

Notice however that, in order to guarantee freedom from the run-time error consisting of assigning a value to a non-existing array element we should also add a proof that

{0 ≤ Counter ≤ MaxLength–1}always holds before any assignement to variable `Text[Counter]`: this can be proved in our case thanks to the fact that within the loop body of the fixed version at the point where `Text[Counter]` is assigned variable Counter has not been changed yet and therefore the loop condition `Counter < MaxLength` still holds.

**Important remark on solution evaluation**

A solution which considered the fragment as correct and prove it vas such without explicitly requiring freedom from the run-time error would be evaluated anyway as “good enough”.

# Analisi e Progetto di Sistemi Critici

# Formal Methods

June 18th, 2008

**NB: the assigned time is 2 hours.**

**The score obtained will be added to the ones obtained in the first midterm test and in the homework to produce the final score of the whole exam.**

A server serves three devices which generate asynchronous requests. Two of them, say D1 and D2, are identical and may generate *no more than* one request every 10 time units. The third one, say D3, is different and generates at most one request every 20 time units.

As soon as it is available the server begins serving any pending request nondeterministically; it takes exactly 2 time units to serve requests from D1 and D2 and 3 time units to serve requests from D3. After serving any request, the server is available again to serve further requests.

Formalize the above system by means of a suitable timed PN. Then prove by means of TRIO formulas and deductions that the system behaves in such a way that *every request will be served.*

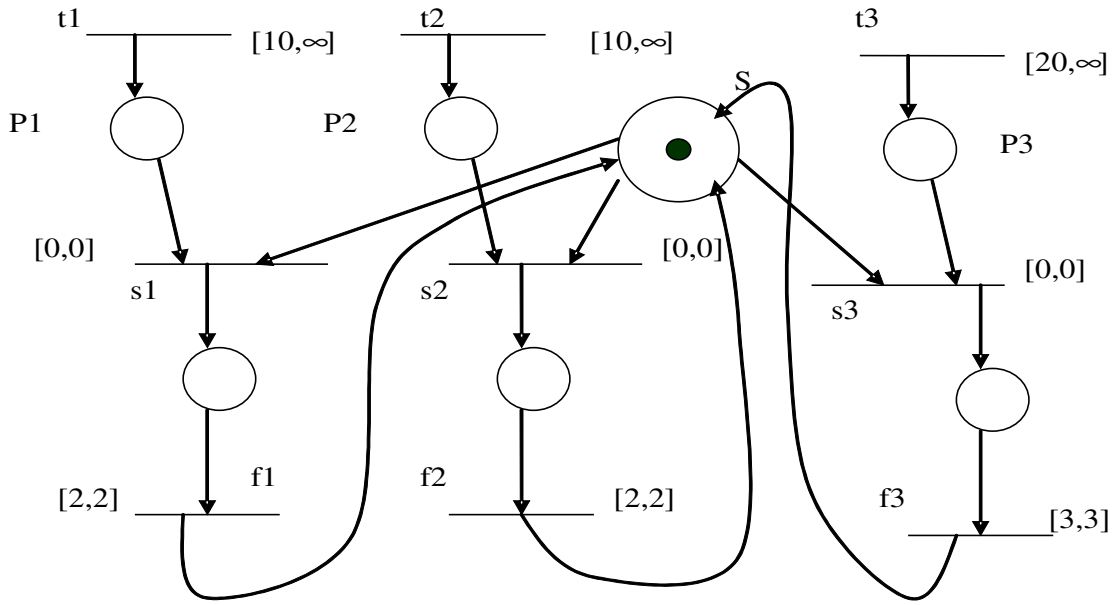
**Optional part (it is *strongly recommended* that the optional part is taken only after completing the basic part).**

Modify the previous formalization in such a way that D3 has priority over D1 and D2, without being allowed to preempt them; in other words: if a request comes from D3 before or simultaneously with other requests, D3 is preferred; if it comes while the server is serving another request, the pending request is first finished and then D3's request is served immediately.

If useful/necessary, you may enrich the basic timed PN formalism in such a way that it can better formalize this new policy. Then, prove that in this case not only any request is served but also that *any request by D3 is served within* k *time units* (specify the value of *k*).

**Outline of a possible solution.**

The following timed PN is a suitable description of the basic system (no priority)




Notice that the above net is not conflict-free. Thus the simplified axiomatization must be extended to deal with conflicts, by stating a mutual exclusion on the firing of s1, s2, s3, e.g. by using a XOR in the UB axioms. Notice also that the net is potentially unbounded but the timing is such that any place will never be more than single-marked. Thus, the axiomatization can be extended only in a way that just the occurrence of a double-marking is formalized and then it is proved that such an event never occurs (see e.g. the exercise proposed in the June 15, 2005 test).

Informally, it is clear that in the worst case the three transitions generating the requests fire together; then, there is time to serve the three of them before anyone can fore again.

To formalize the above reasoning one should state that:

- Initially only S is marked
- If, say, t1 fires when P2 and P3 are not marked, then f1 fires within 2 time units
- If t1 fires when P2 and/or P3 are marked, then in the worst case s1 fires within 5 time units; at that time P2 and P3 are certainly not marked; therefore s1 fires and f1 fires within 2 more time units.
- The firing of t2 and t3 is dealt with similarly.

**Optional part**

A natural way to deal with the optional part is to exploit inhibitor arcs: an inhibitor arc joining P3 with the output transitions of P1 and P2 gives priority to the serving of P3. Then, by exploiting a suitable formalization of the behavior of timed PNs with inhibitor arcs (see e.g. Exercise 1 of the final test of June 2003) the proof proceeds in a similar way as in the basic case.

# Analisi e Progetto di Sistemi Critici: parte Hoare

# Formal Methods: Hoare part

June 17th, 2009

Consider the problem of factorizing a positive integer number as the product of powers of prime numbers: for instance:

$99 = 3^2*11$ ( also = $2^0*3^2*5^0*7^0*11^1$); $250 = 2^1*5^3$; $3528 = 2^3*3^2*7^2$; …

We want to build a program (fragment) that, given a positive integer number n, computes the exponents of the prime numbers factorizing n: for instance, the result corresponding to 99 should be [0,2,0,0,1], etc.

For the sake of simplicity you may assume that:

- n is already stored in the core memory (as usual, we may forget about I/O)
- an array of positive integer, Prime, is also already stored in the memory. Prime contains, in increasing order all prime numbers: 2, 3, 5, 7, 11, 13, …
- Prime is conceptually infinite: you may neglect its bounds
- the program fragment must build an array Exp that stores all powers that should be applied to the prime numbers to obtain n, including the 0's up to the maximum prime number whose exponent is not 0. For instance, in correspondence of 99 Exp should be the above array [0,2,0,0,1].
- For Exp too you may neglect the fact that it is conceptually unbounded and build it only up to the last exponent greater than 0.

With reference to the above problem, do, in the suggested order, the following actions:

1. Give a formal specification of the problem by means of suitable Pre- and Post-conditions.
2. Write a program fragment (by using a typical Pascal-like or C-like minilanguage) that solves the problem.
3. For each loop of your program write an invariant suitable to prove the partial correctness of your program with respect to the given specification. *Briefly sketch* how such a proof could be arranged: at this stage you do not have to fully develop the proof: just give a few lines of explanation of how you could build it: e.g. through suitable lemmas.
4. Briefly sketch how you could build a termination proof. Again, you do not have to fully develop the proof: just provide the essential elements thereof, e.g., the chosen well-founded set(s) and an indication of how the loop body (or bodies) transforms the values within such set(s).

**Optional part to be carried over only after completing the previous items.**

5. Complete the missing details in the partial correctness proof
6. Complete the missing details in the termination proof.

**Hints for the solutions**

**1.**

Possible pre and post conditions:

PRE: $\{n > 1 \wedge \forall\, i\ (i \geq 1 \rightarrow \text{prime}\ (\text{Prime}(i) \wedge (\text{Prime}(i) < \text{Prime}(i+1)))) \wedge$

$\forall\, x\ (\text{prime}\ (x) \rightarrow \exists\, j\ (x = \text{Prime}(j))\}$

/* prime (x) means that x is a prime number; its formal definition is omitted*/

POST: $\{n = \prod_{j=1}^{i-1} (\text{Prime}(j)^{\text{Exp}(j)})\}$

**2.**

A possible Pascal-like program fragment:

```
begin
  i := 1; RIS := n;
  while RIS > 1 do
        k := Prime(i); c := 0;
        while (1 < RIS and RIS mod k = 0) do
              RIS:= RIS div k; c := c +1
        end;
        Exp(i) := c; i := i+1;
  end
end
```

**3 (and part of 5).**

A possible invariant for the external loop is the following:

IEXT: $(n = \prod_{j=1}^{i-1} (\text{Prime}(j)^{\text{Exp}(j)}) * \text{RIS}) \wedge \text{RIS} \geq 1$

In fact IEXT trivially holds after initialization and implies the postcondition at loop exit thanks to the negation of the loop condition.

Backwards propagation of IEXT through Exp(i) := c; i := i+1; yelds

IEXT*: $(n = \prod_{j=1}^{i} (\text{Prime}(j)^{\textbf{if } j = i \textbf{ then } c \textbf{ else } \text{Exp}(j)}) * \text{RIS}) \wedge \text{RIS} \geq 1$

IEXT* itself, with the addition of k = Prime(i), can be assumed as an invariant for the inner loop.

In fact, it obviously implies itself, without the addition; furthermore the clause

$(n = \prod_{j=1}^{i} (\text{Prime}(j)^{\textbf{if } j = i \textbf{ then } c \textbf{ else } \text{Exp}(j)}) * \text{RIS})$ is clearly preserved through the body (c is increased by 1 and RIS is divided by k, i.e. Prime(i)). RIS ≥ 1 is maintained through RIS:= RIS **div** k thanks to the loop condition 1 < RIS **and** RIS **mod** k = 0.

Finally backwards propogation of IEXT* through k := Prime(i); c := 0; is obviously implied by IEXT itself plus RIS > 1.

**4 (and part of 6).**

Clearly, the essential trace of program execution is driven by the values assumed by RIS: in fact RIS takes value over N (a well founded set); the invariant states that it is always ≥ 1; furthermore, at every iteration of the inner loop its value is decreased. Thus, the inner loop certainly terminates.

Proving the termination of the external loop is definitely more intricate. The proof can go along the following lines: consider the difference RIS – Prime(i): it takes values over integers; observe however that actually it is always in N (RIS - Prime(i) ≥ 0 is invariant): in fact at every loop iteration RIS is either prime, and therefore there must exist a value i such that RIS = Prime(i), or it is not and therefore ∃ i, j, h such that RIS = Prime(i)* Prime(j)* h (this is actually an inductive proof that every positive integer can be (uniquely) factorized as the product of powers of prime numbers). At every iteration the difference RIS - Prime(i) is decreased and therefore the loop is eventually exited.

# Analisi e Progetto di Sistemi Critici

# Formal Methods

June 23, 2009

**NB: the assigned time is 2 hours.**

**The score obtained will be added to the ones obtained in the first test (on Hoare's method) and in the homework to produce the final score of the whole exam. Minor adjustments would be possible.**

Consider the following slight generalization of the KRC problem, say it QRC:

Whereas in KRC we consider just one train in one track, we now assume that any number of trains can go through the only track. Trains however are coming at a time distance from each other of at least $\Delta$ time units. All trains are obviously running in the same direction; opening and closing the gate takes as usual $\gamma$ time units and cannot be interrupted; the time taken by the trains to go through the various region are the usual ones ($[d_m, d_M]$, $[h_m, h_M]$).

1. Formalize the new version QRC as a suitable timed PN.
2. Assume that $\Delta > d_M + h_M + \gamma$. Then prove –or disprove!- by means of a suitable TRIO axiomatization that:
    a. The net is still 1-bounded. **NB**: this fact must be *proved* by means of a suitable extension of the "normal axiomatization", it cannot be assumed as an *a priori* hypothesis.
    b. The safety property, i.e., whenever a train is in the critical region the gate is closed, is still guaranteed.

Notice that, in order to *prove* that a net is 1-bounded you must first formalize the fact that more than one token is stored in a place: this just to *express* the fact that such a situation does *not* occur; then the axiom system can be used to prove that such a fact never occurs. You do *not* have to provide *a full axiomatization* suitable to cope with any general net and any possible property: it is sufficient that you provide enough definitions, axioms, and rules so that the above property can be proved.

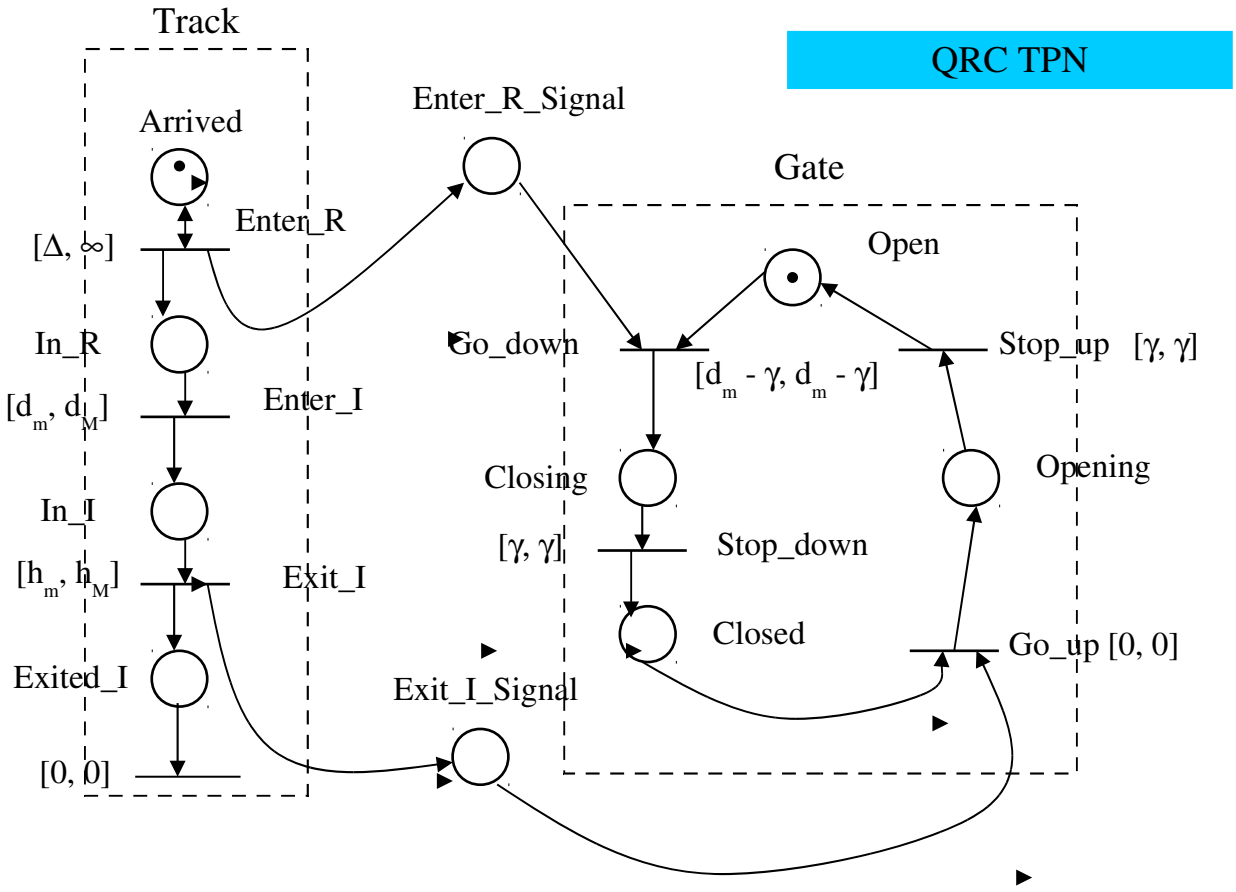
**Optional part (to be done only after having completed the main part!)**

Would it be possible to prove properties a. and/or b. even with a value of $\Delta < d_M + h_M + \gamma$. ?

If so, which constraint on $\Delta$ should be assumed to prove both properties? Should the "policy" of the gate be arranged accordingly? How?

**Sketch for a solution**

**1.** The TPN below, a slight modification of the one formalizing the KRC, enriches it by adding the possibility the trains arrive at a distance in time of at least $\Delta$ between each other.




**2. Hints for a possible axiomatization and proof**

The simplified axiomatization assumes a priori that the net is 1-bounded. Here, in principle, we must also formalize the fact that more than one token is stored in a place: this just to *express* the fact that such a situation does not occur.

If we do not strive for full generality, we can formalize this fact by a formula that states that a place is “doubly-marked” iff sometimes in the past –including the current instant- it was singly-marked and its input transition fired and its output transition did not fire since then, –again, including the current instant-.

With such a new definition, one can easily express the desired property.

The axiom system should be slightly extended to formalize the behavior with multiply marked places. However, since in this case we are only interested in proving that our net will never be doubly marked in any place, it is sufficient to use axioms specifying

e

the behavior of singly marked places and then showing that conditions leading to doubly marking never occur.

At this point the proof of the desired property can be carried over in a fairly similar way as to the normal proof of the KRC. The main difference is that Lemma 3, i.e.,

**Every transition of the net of Figure KRC TPN can fire at most once**

Must be replaced with a different but similar one such as e.g.,

**Every transition of the net of Figure KRC TPN can fire at most once during a time interval < $\Delta$.**

Then, the whole proof of the safety property of the KRC can be arranged for the QRC accordingly (e.g., the sentence "Since, by Lemma 2.B, Go_Down could not fire before a firing of Enter_R, then when Enter_R fired q instants in the past …" becomes "Since, by Lemma 2.B, Go_Down could not fire before a firing of Enter_R *within the last $\Delta$ time interval,* then when Enter_R fired q instants in the past …).

**Optional part**

If $\Delta$ is > $d_M$ and $d_M$ > $h_M$ , (a reasonable but necessary assumption) the no train "accumulates" in the R and in the I region. However, in this case we could have a token in In_I when the gate is still opening, thus violating the safety property. Instead, if $\Delta > d_M + 2.\gamma$, (or $\Delta > d_M + \gamma$,and $d_M - d_m > \gamma$) and the Go_Down command is issued immediately, i.e. with time bounds [0, 0], then, when a new train is in the critical region there has been enough time to complete the opening and close again the gate.

Other more sophisticated options could consist of preventing opening the gate at the arrival of a new train or interrupting the opening. This would require major changes in the PN structure, possibly introducing inhibitor arcs or other changes in the model.