Algèbres de réseaux de Petri

Master 2 CNS/SA
Examen de première session
9 novembre 2019

Durée totale: 2h

Avertissement. Lisez *attentivement* le sujet. Les calculatrices, les documents, et les téléphones portables sont interdits. Vous devez *expliquer et justifier toutes vos réponses*. Si le sujet vous semble comporter des erreurs ou imprécisions, détaillez à l'écrit et répondez de votre mieux. *Il ne sera répondu à aucune question concernant le sujet.* Cet énoncé comporte 5 pages.

Exercice 1 (Train-train)
On considère un modèle de passage à niveau en ABCD:

```
1  symbol RED, GREEN, OPEN, MOVE, CLOSED, DOWN, UP
2
3  buffer light : enum(RED, GREEN) = GREEN
4  buffer command : enum(UP, DOWN) = ()
5
6  net gates () :
7      buffer state : enum(OPEN, MOVE, CLOSED) = OPEN
8      ([command-(DOWN), state-(OPEN), state+(MOVE)] ;
9       [state-(MOVE), state+(CLOSED), light-(RED), light+(GREEN)] ;
10      [command-(UP), state-(CLOSED), state+(MOVE)] ;
11      [state-(MOVE), state+(OPEN)])
12     * [False]
13
14 net track () :
15     buffer crossing : bool = False
16     ([command+(DOWN), light-(GREEN), light+(RED)] ;
17      [light?(GREEN), crossing-(False), crossing+(True)] ;
18      [crossing-(True), crossing+(False), command+(UP)])
19     * [False]
20
21 gates() | track()
```

Sa sémantique est donnée par le réseau de Petri la figure 1 page suivante que vous devrez compléter. Écrivez directement sur l'énoncé et rendez-le avec votre copie.

Questions.

1. Identifiez par un e les places d'entrée. (*1 point*)
2. Identifiez par un x les places de sortie. (*1 point*)
3. Identifiez par leurs noms les places de buffer light, command, gates().state, et track().crossing. (*1 point*)
4. Associez chaque transition à l'action correspondante en inscrivant dans la transitions le numéro de la ligne à laquelle l'action est définie. (*2 points*)
5. Étiquetez les arcs entre les places de buffer et les transitions, sur les arcs aller-retour, écrivez x/y si x est consommé et y est produit à sa place. (*3 points*)

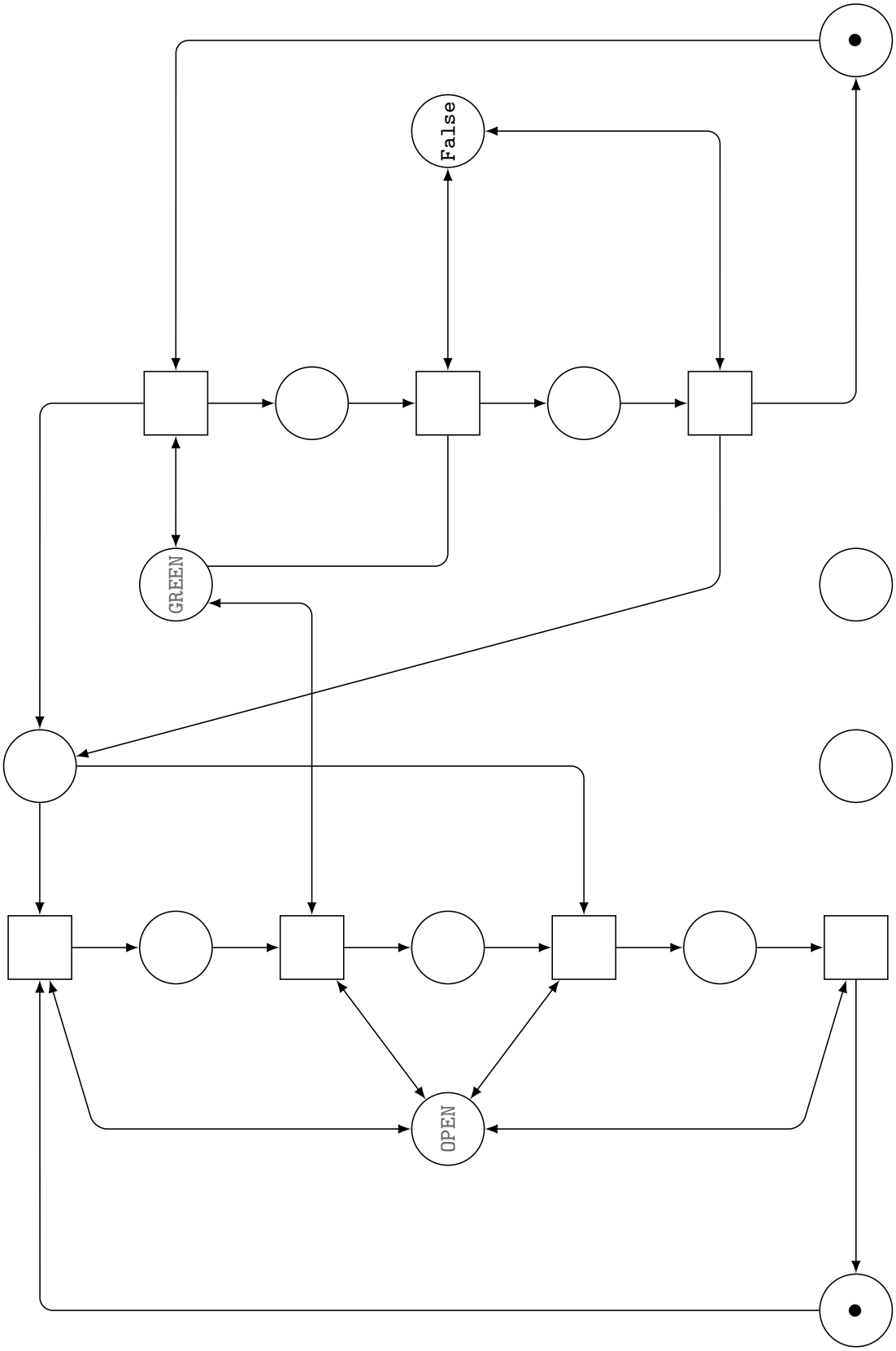
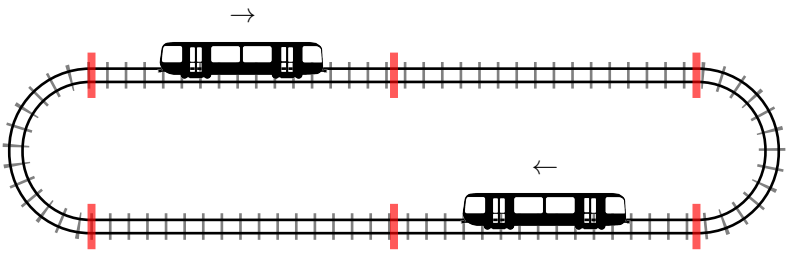


Fig. 1. Réseau de Petri à compléter pour l'exercice 1.

Exercice 2 (Métro, exo, dodo)
On considère une ligne de métro automatique comme une seule voie circulaire.



On suppose le fonctionnement suivant:

- la ligne comporte t tronçons (dans le dessin ci-dessus, il y en a 6 séparés par des lignes rouges);
- chaque tronçon sauf les deux aux extrémités de la ligne contient une station en son milieu;
- on ne modélisera pas l'arrêt aux stations;
- il doit toujours y avoir au moins un tronçon vide entre deux rames;
- les rames gèrent l'extinction et l'allumage du courant dans les tronçons de façon à garantir cette condition;
- pour qu'une rame dans un tronçon puisse avancer, il faut que le tronçon suivant soit allumé;
- dès qu'un rame atteint un nouveau tronçon, elle coupe le courant dans celui qu'elle vient de quitter, et rallume le courant dans celui qui se trouve encore avant.

Questions.

1. Compte tenu de ce protocole, combien de rames au maximum peut contenir une ligne de métro qui comporte t tronçons? (*2 points*)
2. Proposez une façon d'identifier les tronçons de façon à pouvoir calculer facilement les identifiants des deux tronçons qui précèdent un tronçon donné. (*2 points*)
3. Proposez une façon de représenter dans un buffer ABCD l'état d'allumage de chaque tronçon. (*2 points*)
4. Proposez une modélisation d'une rame par un net ABCD, paramétré par sa position initiale (c'est-à-dire par un identifiant de tronçon). (*4 points*)
5. Modélisez le système dessiné ci-dessus, avec ses deux rames et l'état initial de ses tronçons. (*2 points*)

ABCD cheatsheet

```
$ abcd [option]...  spec.abcd
```

```
# a comment
# another comment
declarations
process
```

`--pnml=FILE` save net as PNML (SNAKES' variant)

`--dot=FILE` `--neato=FILE` `--towpi=FILE`
`--circo=FILE` `--fdp=FILE` draw net using a GraphViz engine

`--load=PLUGIN` load a plugin before to build net (may be repeated)

`--simul` start interactive simulator

buffer declarations:
`buffer name: type = ()`
▷ empty buffer
`buffer name: type = val, ...`
▷ buffer with initial content

type expressions:
any Python type or class
▷ eg, `int`, `str`, `object`, ..., or user-defined classes
`enum(val, val, ...)`
▷ enumerated type
`type * type`
▷ cross-product of types
`type | type`
▷ union of types
`type & type`
▷ intersection of types

types definition:⋆
`typedef name: type`

constants:⋆
`const name = expr`

symbols:⋆
`symbol name, ...`
▷ define fresh unique values

Python imports:⋆
▷ just use regular Python imports

sub-processes

⋆ global declarations only

control flow:
`process ; process`
▷ sequential composition
`process + process`
▷ non-deterministic choice (use opposite guards to make it deterministic)
`process * process`
▷ iterate left-hand-side and exit with right-hand-side
`process | process`
▷ parallel composition

(no priorities ⇒ use parentheses)

sub-process instances:
`name(args)`
▷ substitute `args` in net `name`
scope its local buffers
insert the net

sub-process declarations:
`net name (params):`

indented block
declarations
process

sub-process parameters:
a comma-separated list of:
`name`
▷ a value is expected
`name: buffer`
▷ a buffer name is expected

atomic actions:
`[True]`
▷ no-op non-blocking action
`[False]`
▷ always-blocking action
`[accesses]`
▷ unguarded action
`[accesses if expr]`
▷ guarded action

accesses:
`buff-(val)`
▷ consume `val` from `buff`
`buff-(var)`
▷ consume a value from `buff` and binds it to `var`
`buff+(expr)`
▷ produce a value into `buff`
`buff?(val)`
▷ test for `val` in `buff`
`buff?(var)`
▷ test for a value in `buff` and binds it to `var`
`buff>>(var)`
▷ flush `buff` into `var`
`buff<<(var)`
▷ add the values contained in `var` to `buff`
`buff<>(val=expr)`
▷ replace `val` in `buff` with `expr`
`buff<>(var=expr)`
▷ replace `var` in `buff` with `expr`

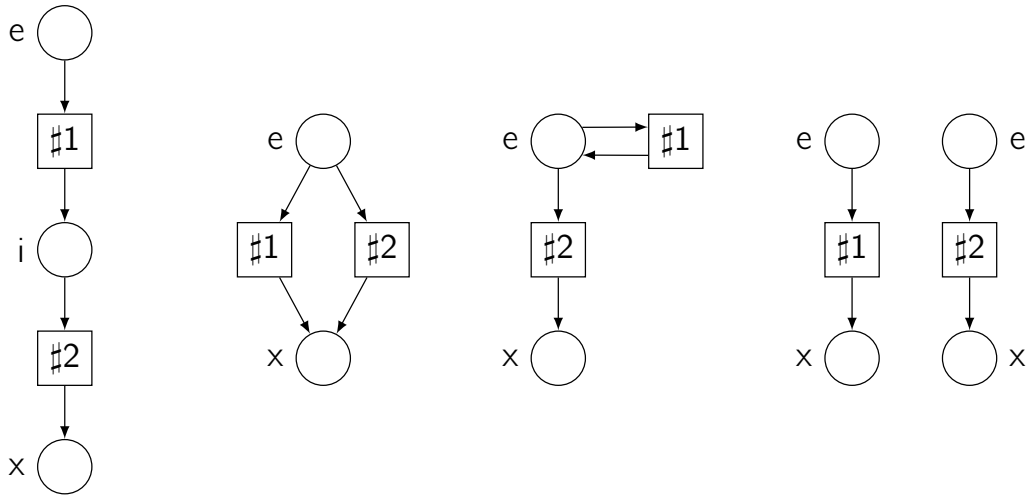
(a comma-separated list of accesses is performed atomically)

© 2018 Franck Pommereau
franck.pommerau@univ-evry.fr
snakes.ibisc.univ-evry.fr

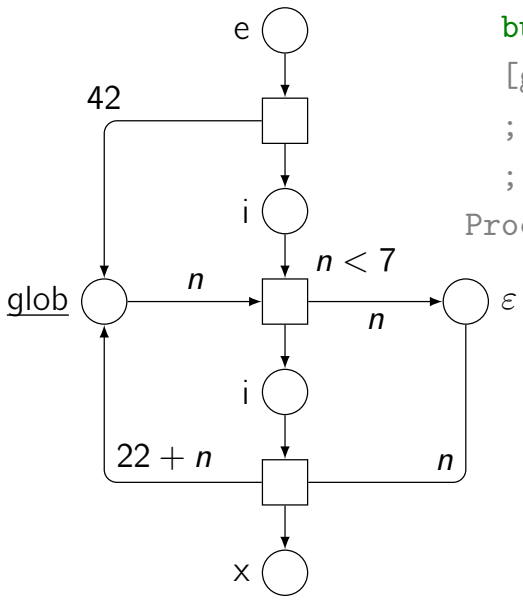
Specifying control-flow operators

with operator nets

sequence ⨾ choice □ iteration ⊛ parallel ||



Translating ABCD into Petri nets




```
buffer glob :  int = ()
net Process (num=22) :
  buffer loc :  int = ()
  [glob+(42)]
  ; [glob-(n), loc+(n) if n < 7]
  ; [loc?(n), glob+(22 + n)]
Process(22)
```