Algèbres de réseaux de Petri

Master 2 CNS/SA
Examen de première session
14 novembre 2023

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. Cet énoncé comporte 4 pages.

Exercice 1 (ABCD)

```
buffer foo : int = ()
buffer bar : int = 0

net spam (s) :
    [bar-(x), bar+(x+s), foo+(x)]
    * [False]

spam(1) | spam(2) | spam(3) | spam(4)
```

1. Expliquez pourquoi on ne peut pas analyser ce modèle par des techniques de *model-checking* se basant sur le calcul de l'espace d'états. (*2 points*)
2. Prouvez que le modèle peut atteindre un état dans lequel le buffer `bar` contient la valeur 812. (*4 points*)
3. Prouvez que le modèle ne peut jamais atteindre un état dans lequel le buffer `bar` contient simultanément la valeur n et la valeur $n + 42$, pour tout $n \geq 0$. (*4 points*)

Suite sur la page suivante...

Exercice 2 (Uno)
On souhaite modéliser une version simplifiée du jeu Uno. Il se joue avec des cartes, chacune portant un numéro de 0 à 9 et une couleur parmi rouge, jaune, vert, et bleu. Le jeu se déroule ainsi :

(1) Le paquet de cartes, mélangé et posé faces cachées, constitue la pioche.

(2) On retourne une carte de la pioche pour constituer le talon.

(3) Au démarrage, chaque joueur à son tour pioche 7 cartes.

(4) Puis les tours de jeu commencent. Chaque joueur joue à son tour :

  (a) s'il possède une carte de la même couleur ou portant le même numéro que le talon, il la pose dessus, face visible, et elle devient le nouveau talon ;

  (b) s'il ne possède aucune carte qui correspond au talon, il pioche une carte.

(5) Le jeu continue en répétant l'étape (4) jusqu'à ce que :

  (a) un joueur n'a plus de carte, alors il gagne ;

  (b) personne ne peut plus jouer, alors le gagnant est celui qui a le moins de cartes (ou dont la somme des cartes est la plus basse en cas d'égalité).

Le but de l'exercice est de modéliser ce jeu en ABCD, en répondant progressivement aux questions ci-dessous. Pour cela, on suppose que pour chaque couleur, il y a exactement une carte portant chaque numéro (donc 4×10 cartes). De plus, on ne modélisera pas la fin de partie décrite à l'étape (5). Enfin, un joueur sera modélisé par un net paramétré par son numéro (entre 0 et $n-1$) et déclarant un buffer `main` contenant ses cartes.

Questions :

1. Proposez une modélisation simple pour les cartes qu'on pourra rassembler dans un buffer `pioche` (les cartes n'y étant pas ordonnées, on n'aura pas besoin de modéliser le mélange). (*1 point*)
2. Proposez une modélisation simple du talon de façon à minimiser son nombre de configurations. (*1 point*)
3. En supposant que le premier joueur a le numéro 0. Proposez une façon simple de modéliser le tour des joueurs. (*1 point*)
4. Écrivez un net pour initialiser le jeu comme dans les étapes (1) et (2) et pour initialiser le tour des joueurs. (*2 points*)
5. Expliquer pourquoi, afin de réaliser l'action (4.b), il est nécessaire de pouvoir disposer de tous les jetons du buffer `main`. Expliquez comment récupérer tous ces jetons peut être réalisé en ABCD. On suppose qu'on dispose d'une fonction `play(t,M)` qui renvoie une carte parmi la main `M` si elle est jouable sur le talon `t`, ou bien renvoie `None` si aucune carte n'est jouable. Indiquez de quels types vous avez besoin pour `M` et `t` selon ce que vous aurez répondu plus haut. (*1 point*)
6. Écrivez un modèle du joueur incluant les étapes (3) et (4), l'étape (4) étant répétée indéfiniment. (*3 points*)
7. Écrivez un modèle du jeu avec 4 joueurs. (*1 point*)

ABCD cheatsheet

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

`--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

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

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

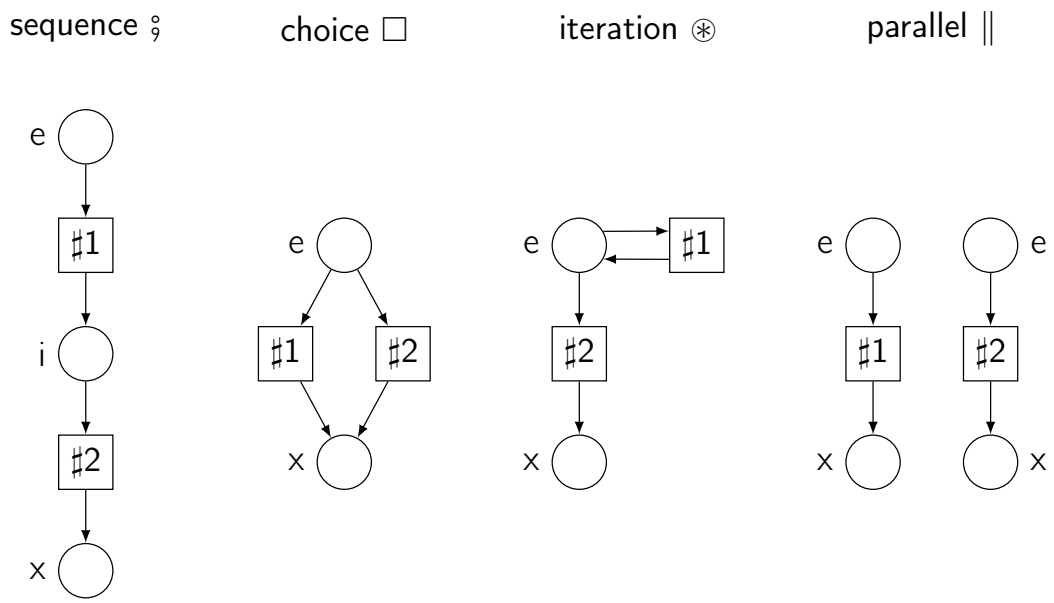
atomic actions:
`[True]`
▷ no-op non-blocking action
`[False]`
▷ always-blocking action
`[accesses]`
▷ unguarded action
`[accesses if expr]`
▷ guarded action

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`

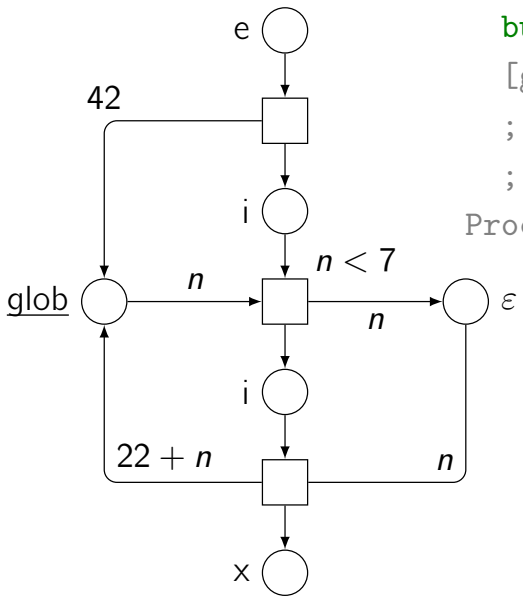
(a comma-separated list of accesses is performed atomically)

Specifying control-flow operators

with operator nets



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)
```