Conception et vérification

Master 2 CILS
Examen de première session
14 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. Ce sujet comporte 2 pages.

Exercice 1
On se donne le modèle ABCD suivant (l'opération "`x % y`" est le *modulo* et calcule donc le reste de la division entière de `x` par `y`) :

```
buffer foo : int = range(2, 1000)  # tous les entiers entre 2 et 9999 inclus
buffer bar : int = ()

[foo-(x), foo?(y), bar+(x) if x % y == 0] * [False]
```

Questions:

1 • Expliquez pourquoi ce système ne peut pas avoir d'exécution infinie. (*2 points*)

2 • Expliquez les changements de contenus des buffers à chaque itération, regardez en particulier quelle est la caractéristique de `x` d'un point de vue arithmétique. (*2 points*)

3 • Expliquez la caractéristique commune des jetons dans chacun des buffers à la fin de l'exécution. (*2 points*)

Exercice 2
On considère le protocole de Denning-Sacco pour distribuer une clef partagée via un serveur de confiance :

$$\begin{aligned} A \to S &: A, B \\ S \to A &: \langle B, K_{A,B}, T, \langle K_{A,B}, A, T\rangle_{K_{B,S}}\rangle_{K_{A,S}} \\ A \to B &: \langle K_{A,B}, A, T\rangle_{K_{B,S}} \end{aligned}$$

où A et B sont les identités des agents Alice et Bob, S est l'identité du serveur, $K_{x,y}$ est une clef secrète partagée par x et y, et T est un horodatage (*timestamp*).

Questions:

1 • Donnez deux propriétés attendue de l'horodatage permettant d'assurer que (1) les messages ne sont jamais réutilisé, et (2) un message peut être identifié comme nouveau. (*1 point*)

2 • Proposez une modélisation symbolique de l'horodatage permettant d'assurer ces propriétés. (*1 point*)

3 • Proposez une modélisation ABCD de ce protocole, sans attaquant. (*5 points*)

Exercice 3
On souhaite modéliser un distributeur automatique de billets. Pour simplifier, on ne modélisera pas les sommes, mais uniquement les éléments suivants :

— l'utilisateur doit entrer un code PIN ;

— la carte est capturée après trois mauvais codes PIN ;

— la banque est interrogée pour indiquer si le retrait est autorisé ou pas.

Questions:

1 • Donnez dans un tableau les acteurs, leurs états, et les actions. (*3 points*)

2 • Proposez une modélisation ABCD de ce système. (*4 points*)

ABCD cheatsheet

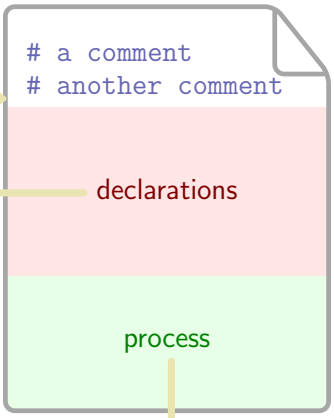
```
$ abcd [option]...  spec.abcd
```

`--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



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)