Conception et vérification

Master 2 CILS
Examen de première session
12 novembre 2018

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 considère le modèle ABCD suivant :

```
1 buffer S : int = ()
2 buffer L : int = ()
3 net Foo (a, b) :
4     [S+(a), L+(b) if a<b]
5     + [S+(b), L+(a) if a>b]
6     + [S+(a), S+(b), L+(a), L+(b) if a==b]
7 Foo(1,2) ; Foo(4,3) ; Foo(2,3)
```

Questions :

1. Expliquez pourquoi ce modèle n'a qu'une exécution possible. (*2 points*)
2. Donnez l'état des buffers à la fin de cette exécution. (*2 points*)
3. Expliquez pourquoi remplacer les opérateurs "`;`" dans la ligne 7 par des "`|`" change la réponse à la question 1 mais pas celle à la question 2. (*3 points*)

Exercice 2
On considère le protocole *Wide Mouthed Frog* dont le but est d'échanger une clef de session fraîche par l'intermédiaire d'un serveur de confiance.

(1)	$A \rightarrow S : A, \{T_a, B, K_{ab}\}_{K_{as}}$	A transmet la clef à S, à l'attention de B
(2)	$S \rightarrow B : \{T_s, A, K_{ab}\}_{K_{bs}}$	S envoie la clef à B, de la part de A

Dans ce protocole : A et B sont des agents, S est le serveur, T_x est un *timestamp*, c'est-à-dire une date (et donc, si T_x et T_y sont deux timestamps successifs, alors $T_x < T_y$; de plus les timestamps sont des valeurs fraîches mais pas forcément imprévisibles), et K_{xy} est une clef secrète partagée par X et Y.

Questions :

1. Expliquez comment traiter de façon symbolique les timestamps. Expliquez en particulier comment faire en sorte qu'ils soient toujours différents et ordonnés. (*3 points*)
2. Donnez une modélisation en ABCD de A. (*2 points*)
3. Donnez une modélisation en ABCD de B. (*2 points*)
4. Donnez une modélisation en ABCD de S. (*2 points*)
5. Donnez une modélisation en ABCD de l'attaquant. Expliquez en particulier ses connaissances initiales. (*1 point*)
6. Donnez une modélisation en ABCD d'un scénario permettant deux sessions successives entre A et B par l'intermédiaire de S. (*3 points*)

ABCD cheatsheet

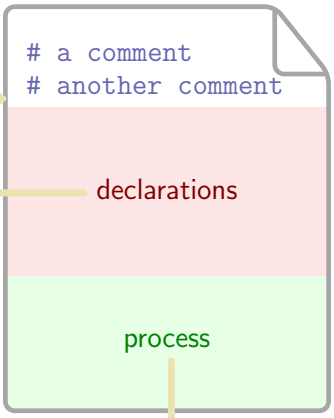
```
$ abcd [option]...  spec.abcd
```

`--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)

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