State Sequences

- A state is a type-consistent interpretation of the system variables V
- A state sequence is an infinite sequence of states, represented as a mapping from time ($\mathbb{N}$) to states:

  STATE_SEQ: TYPE = [TIME $\longrightarrow$ STATE]

  (Recall: $\sigma : s_0, s_1, s_2, \ldots$)
- Assertions are properties defined on individual states, without reference to their position in the state sequence.

  ASSERTION: TYPE = [STATE $\longrightarrow$ bool]

  Disjunction, conjunction, negation and implication over assertions are defined in the natural manner.

Temporal Operators

Let S be a state sequence.

We denote the notion of a temporal property p holding at position $j \geq 0$ of S by $p(S, j)$.

If p is an assertion then $p(S, j) = p(S(j))$

For the LTL operators :

Henceforth, $\Box$, G	$G(p)(S, j)$	$\Longleftrightarrow$	$p(S, k)$ for all $k \geq j$
Eventually, $\Diamond$, F	$F(p)(S, j)$	$\Longleftrightarrow$	$p(S, k)$ for some $k \geq j$
Next, $\bigcirc$, X	$X(p)(S, j)$	$\Longleftrightarrow$	$p(S, j + 1)$

Example

$V = \{c : \{ \text{RED}, \text{BLUE}\} \}$
Two distinct states: $\langle c : \text{RED}\rangle$, $\langle c : \text{BLUE}\rangle$
`is_blue` : `ASSERTION` $= (\lambda\ (s$: `STATE`$)$: $s`c = \text{BLUE})$

Consider state sequence SS : [`TIME` $\mapsto$ `STATE`] defined by the first 2 columns in the table:

$time$	$state$	`is_blue`	$X($`is_blue`$)$	$F($`is_blue`$)$
0	$\langle c : \text{BLUE}\rangle$	T	F	T
1	$\langle c : \text{RED}\rangle$	F	F	T
2	$\langle c : \text{RED}\rangle$	F	T	T
3	$\langle c : \text{BLUE}\rangle$	T	T	T
4	$\langle c : \text{BLUE}\rangle$	T	T	T
5	$\langle c : \text{BLUE}\rangle$	T	T	T

$\cdots$

- Assertion `is_blue` depends only on the state. `is_blue`(SS, i) = `is_blue`(SS(i))
- Temporal properties depend on the whole state sequence. $SS(0) = SS(3)$ but X(`is_blue`)$(SS, 0)$ is T, X(`is_blue`)$(SS, 3)$ is F
- X(`is_blue`)$(SS(0))$ is incorrectly typed as $SS(0)$ is the state $\langle c : \text{BLUE}\rangle$ (as is $SS(3)$!!)

FDS - Fair Discrete System

Recall, FDS $D = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$ consists of :

- V : a set of typed state variables. A V-state is an interpretation of V. Σ_V is the set of all V-states.
- Θ : The initial condition. An assertion characterizing the initial states.
- ρ : The transition relation. A predicate $\rho(V, V')$ referring to the both unprimed (current) and primed (next) versions of state variables.
- $\mathcal{J}$: The set of justice requirements. Each computation must have infinitely many J_i-states, for every $J_i \in \mathcal{J}$.
- $\mathcal{C}$: The compassion requirements, each of the form $\langle p, q\rangle$. Infinitely many p-states imply infinitely many q-states.

PFS - Parameterized Fair System

```
PFS: TYPE =
 [# initial: ASSERTION,
    rho: BI_ASSERTION,
    justice: JUSTICE_TYPE,
    compassion: COMPASSION_TYPE #]
```

where

```
BI_ASSERTION: TYPE = [STATE, STATE → boolean]

JUSTICE_TYPE: TYPE = [TRANSITION_DOMAIN → ASSERTION]

COMPASSION_PAIR: TYPE =
    [# p: ASSERTION, q: ASSERTION #]

COMPASSION_TYPE: TYPE =
    [TRANSITION_DOMAIN → COMPASSION_PAIR]
```

Note:

- STATE and TRANSITION_DOMAIN parameters are given in defining the PFS
- There is no state-variables (V) component

Runs and Computations

A STATE_SEQ seq of pfs is an initialized run if it satisfies

- Initiality : seq(0) is initial i.e. pfs‘initial(seq(0))
- Consecution : For every $t = 0, 1, 2,$ state seq$(j+1)$ is a successor of seq(t). I.e. pfs‘rho(seq(t), seq$(t+1)$)

A computation is an initialized run which also satisfies the fairness requirements of:

- Justice: For every $t \in$ TRANSITION_DOMAIN there are infinitely many states in seq at which pfs‘justice(t) holds.
- Compassion: For every $t \in$ TRANSITION_DOMAIN, if there are infinititely many (pfs‘compassion(t)‘p)-states in seq then there are infinititely (pfs‘compassion(t)‘q)-states in seq.

Validity

A temporal property p is termed

- valid if it hold in the first state of every state sequence `seq`.

  is_valid(p)

- P-valid if it hold in the first state of every computation `seq` of program P.

  Assuming that PFS defines program P,

  is_P_valid(p, pfs)

- P-reachable valid if it hold in the first state of every initialized run `seq` of program P.

  is_P_reachable_valid(p, pfs)

Validity ctd

- state sequence $\supseteq$ initialized runs $\supseteq$ computations,

  and so

  validity $\rightarrow$ P-reachable validity $\rightarrow$ P-validity

- Generally, interested in P-validity

- Rules like `BINV` actually prove the stronger P-reachable validity property

- Can always convert P-reachable validity to P-validity.

  Sometimes having the stronger property is useful.

Example: MUX-SEM

$$
\begin{array}{lcll}
\textbf{in} & N & : & \textbf{integer where } N > 1 \\
\textbf{local} & y & : & \{0,1\} \textbf{ where } y = 1
\end{array}
$$

$$
\overset{N}{\underset{p=1}{\|}} P[p] :: \left[\begin{array}{l}
\ell_0 : \textbf{ loop forever do} \\
\quad \left[\begin{array}{ll}
\ell_1 : & \textbf{noncritical} \\
\ell_2 : & \textbf{request } y \\
\ell_3 : & \textbf{critical} \\
\ell_4 : & \textbf{release } y
\end{array}\right]
\end{array}\right]
$$

Figure 1: Parameterized MUX-SEM

A PFS for MUX-SEM

```
muxsem[N: posnat]: THEORY
BEGIN

  IMPORTING more_nat_types

  LOCATION: TYPE = upto[4]

  PROC_ID: TYPE = upto_nz[N]

  TRANS_DOMAIN: TYPE =
     [# loc: LOCATION, pid: PROC_ID #]

  STATE: TYPE =
     [# y: upto[1], loc: [PROC_ID → LOCATION] #]

  IMPORTING PFS[STATE, TRANS_DOMAIN]

  p: VAR PROC_ID
```

```
rho: BI_ASSERTION =
  (λ (current, next: STATE):
    next = current ∨
    (∃ p:
      loc(current)(p) = 0 ∧
      y(next) = y(current) ∧
      loc(next) = loc(current) WITH [(p) := 1]
      ∨
      loc(current)(p) = 1 ∧
      y(next) = y(current) ∧
      loc(next) = loc(current) WITH [(p) := 2]
      ∨
      loc(current)(p) = 2 ∧
      y(current) = 1 ∧
      y(next) = 0 ∧
      loc(next) = loc(current) WITH [(p) := 3]
      ∨
      loc(current)(p) = 3 ∧
      y(next) = y(current) ∧
      loc(next) = loc(current) WITH [(p) := 4]
      ∨
      loc(current)(p) = 4 ∧
      y(next) = 1 ∧
      loc(next) = loc(current) WITH [(p) := 0]))
```



```
st: VAR STATE
t: VAR TRANS_DOMAIN

justice: JUSTICE_TYPE =
  (λ t: (λ st:
    IF loc(t) = 0 ∨ loc(t) = 3 ∨ loc(t) = 4
    THEN loc(st)(pid(t)) ≠ loc(t)
    ELSE TRUE
    ENDIF))

compassion: COMPASSION_TYPE =
  (λ t:
    IF loc(t) = 2
    THEN
      (# p := (λ st: loc(st)(pid(t))=2 ∧ y(st)=1),
         q := (λ st: loc(st)(pid(t)) = 3) #)
     ELSE (# p := (λ st: TRUE), q := (λ st: TRUE) #)
     ENDIF)

pfs: PFS =
  (# initial := {st| y(st)=1 ∧ (∀p: loc(st)(p)=0) },
     rho := rho,
     justice := justice,
     compassion := compassion #)

END muxsem
```

Transition domains

- Very often, as was the case in MUX-SEM, the transition domain is comprised of a location and processor identifier field.

- TRANS_DOMAIN theory, which defines such a transition domain.

- Importing TRANS_DOMAIN[$progSize$, N] creates and imports the following definitions:

```
LOCATION: TYPE = upto[progSize − 1]

PROC_ID: TYPE = upto_nz[N]

TRANS_DOMAIN: TYPE =
   [# loc: LOCATION, pid: PROC_ID #]
```

- In MUX-SEM, we could have defined

```
IMPORTING TRANS_DOMAIN[5, N]
```


Example: BAKERY

$$
\begin{array}{lll}
\textbf{in} & N & : \ \textbf{integer where } N > 1 \\
\textbf{local} & y & : \ \textbf{array } [1..N] \textbf{ of natural where } y = 0 \\
\end{array}
$$

$$
\overset{N}{\underset{p=1}{\Vert}} P[p] :: \left[\begin{array}{l}
\textbf{loop forever do} \\
\left[\begin{array}{ll}
\ell_0 : & \textsf{NonCritical} \\
\ell_1 : & y[p] := \textbf{choose } m \textbf{ such that} \\
 & \quad \forall q : (m > y[q]) \\
\ell_2 : & \textbf{await } \forall q : (y[q] = 0 \vee y[p] < y[q]) \\
\ell_3 : & \textsf{Critical} \\
\ell_4 : & y[p] := 0
\end{array}\right]
\end{array}\right]
$$

Figure 2: Parameterized mutual exclusion algorithm BAKERY

PFS for BAKERY

```
bakery_definition[N: posnat]: THEORY
BEGIN

IMPORTING TRANS_DOMAIN[5, N]

STATE: TYPE =
  [# y: [PROC_ID → nat],
     loc: [PROC_ID → LOCATION] #]

IMPORTING PFS[STATE, TRANS_DOMAIN]

p, q: VAR PROC_ID
```



```
rho: BI_ASSERTION =
 (λ (current, next: STATE):
   next = current ∨
   (∃ p:
     loc(current)(p) = 0 ∧
     y(next) = y(current) ∧
     loc(next) = loc(current) WITH [(p) := 1]
     ∨
     loc(current)(p) = 1 ∧
     (∃ (m: nat): (∀ q: y(current)(q) < m) ∧
     y(next) = y(current) WITH [(p) := m])
     ∧ loc(next) = loc(current) WITH [(p) := 2]
     ∨
     loc(current)(p) = 2 ∧
     (∀ q: q ≠ p → y(current)(q) = 0 ∨
               y(current)(p) ≤ y(current)(q))
     ∧ y(next) = y(current)
     ∧ loc(next) = loc(current) WITH [(p) := 3]
     ∨
     loc(current)(p) = 3
     ∧ y(next) = y(current)
     ∧ loc(next) = loc(current) WITH [(p) := 4]
     ∨
     loc(current)(p) = 4
     ∧ y(next) = y(current) WITH [(p) := 0]
     ∧ loc(next) = loc(current) WITH [(p) := 0]))
```

```
st: VAR STATE

t: VAR TRANS_DOMAIN

justice: [TRANS_DOMAIN → ASSERTION] =
  (λ t: (λ st:
    IF loc(t) = 1
      THEN loc(st)(pid(t)) ≠ 1
        ∨ ¬ (∃ (m: nat): ∀ p: y(st)(p) < m)
    ELSIF loc(t) = 2
      THEN loc(st)(pid(t)) ≠ 2 ∨
        ¬ (∀ q: q ≠ pid(t) →
            y(st)(q) = 0 ∨ y(st)(pid(t)) ≤ y(st)(q))
    ELSIF loc(t) = 3 ∨ loc(t) = 4
      THEN loc(st)(pid(t)) ≠ loc(t)
    ELSE TRUE
    ENDIF))

pfs: PFS =
  (# initial := {st| ∀p: y(st)(p)=0 ∧ loc(st)(p)=0},
     rho := rho,
     justice := justice,
     compassion := empty_compassion #)

END bakery_definition
```


Proving properties of BAKERY

```
yZero: ASSERTION =
   λ st:
      LET y = y(st), loc = loc(st) IN
        ∀ (i: PROC_ID):
          (y(i) = 0 IFF (loc(i) = 0 ∨ loc(i) = 1))

yZero: LEMMA is_P_reachable_valid(G(yZero), fds)
```