Università degli Studi di Udine

Master degree in Artificial Intelligence and Cybersecurity

Fair Transition Systems - Part 3

Angelo Montanari

An example of reactive program

```
local x: integer where x=1

P1:: [ l0: [ l0a: await x=1        ]  ]  ||  P2:: [ m0: loop forever do  ]
     [     [      OR               ]  ]           [     m1: x:=-x        ]
     [     [ l0b: skip             ]  ]           [                      ]
     [ l1:                            ]
```

Properties of the program

- $P2$ never ends.
- Are there computations where $P1$ does not terminate its execution?
  - The **only candidate computation** is:
    $\sigma = < \pi = \{l_o, m_0\}, x = 1 > \rightarrow^{m_0}$
    $< \pi = \{l_o, m_1\}, x = 1 > \rightarrow^{m_1}$
    $< \pi = \{l_o, m_0\}, x = -1 > \rightarrow^{m_0}$
    $< \pi = \{l_o, m_1\}, x = -1 > \rightarrow^{m_1}$
    $< \pi = \{l_o, m_0\}, x = 1 > \rightarrow^{m_0}$ ...
  - **It is not an admissible computation**: the requirement of justice is satisfied by l_{0a}, since it is not constantly enabled, but not by l_{0b}, which is constantly enabled and never executed.

An example of reactive program - variant 1

```
              local x: integer where x=1
P1:: [ ℓ0: [ ℓ0a: await x=1   ] ]  ||  P2:: [ m0: loop forever do ]
       [     [      OR          ] ]          [       m1: x:=-x     ]
       [     [ ℓ0b: await x<>1  ] ]          [                     ]
       [ ℓ1:                      ]
```

Properties of the program - variant 1

- $P2$ never ends.
- Are there computations where $P1$ does not terminate its execution?
  - The **only candidate computation** is the previously identified one.
  - In such a case, it turns out to be an **admissible computation**: the requirement of justice is satisfied both by l_{0a} and by l_{0b}, as no one of them is constantly enabled.

An example of reactive program - variant 2

local x: integer where x=1

$$
P1:: \left[\begin{array}{l} \ell_0\text{: if x=1 then} \\ \quad \ell_1\text{: skip} \\ \quad \text{else} \\ \quad \ell_2\text{: skip} \\ \ell_3\text{:} \end{array} \right] \; \Bigg\| \; P2:: \left[\begin{array}{l} m_0\text{: loop forever do} \\ \quad m_1\text{: x:=-x} \end{array} \right]
$$

Properties of the program - variant 2

- $P2$ never ends.
- Are there computations where $P1$ does not terminate its execution?
  - The **candidate computations** are those where, from a certain point onwards, l_0 (respectively, l_1, l_2) constantly belongs to the value of π.
  - **They are not admissible computations**:
    (the transition corresponding to) the instruction `if` is always enabled and thus sooner or later it must be executed;
    if it is never executed, then the condition of justice is violated,
    if it is executed, then (the transition corresponding to) one of the two instructions `skip` is reached, and, since it is constantly enabled, sooner or later it must be executed;
    if it is never executed, then the condition of justice is violated.

The mutual exclusion problem: the SPL program MUX_SEM

local y: integer where y=1

```
P1:: [ ℓ0: loop forever do              ]  ||  P2:: [ m0: loop forever do              ]
     [      [ ℓ1: noncritical ]         ]  ||       [      [ m1: noncritical ]         ]
     [      [ ℓ2: request y   ]         ]  ||       [      [ m2: request y   ]         ]
     [      [ ℓ3: critical    ]         ]  ||       [      [ m3: critical    ]         ]
     [      [ ℓ4: release y   ]         ]  ||       [      [ m4: release y   ]         ]
     [ ℓ5:                              ]  ||       [ m5:                              ]
```

Properties: mutual exclusion

- **Mutual exclusion**: no computation of the program may include a state where both processes are within their critical region ($P1$ at $l3$ and $P2$ at $m3$).
- If $P1$ is at $l3$, the instruction $l2$ has assigned the value 0 to y (by decrementing it by 1) and, until instruction $l4$ is executed, any attempt of $P2$ to execute $m2$ is bound to fail (as the enabling condition $y > 0$ is not satisfied). A similar argument holds in case $P2$ is at $m3$.

  It is worth pointing out that process $P1$ cannot stay forever in the critical region, that is, it cannot remain indefinitely at position $l3$, thanks to the justice requirement for instruction `critical` (the same remark applies to $P2$).

Properties: accessibility - 1

- **Accessibility**: every state of a computation where a process is at the end of its non-critical region, that is, $P1$ is at $l2$ (resp., $P2$ is at $m2$), must be followed by a state where it is within its critical region, that is, it must be followed by a state where $P1$ is at $l3$ (resp., $P2$ is at $m3$).
- Let us show now that every computation where this is not the case is not admissible.

  Let us assume that $P1$ is blocked at $l2$ (the case in which $P2$ is blocked at $m2$ is analogous). If, from a certain point onwards, y remains constantly equal to 1, that is, $P2$ remains indefinitely within its non-critical region, $l2$ is constantly enabled and never executed, and thus the requirement of justice, which is implied by the requirement of compassion, is violated.

Properties: accessibility - 2

- If y does not remain constantly equal to 1, the computation necessarily has the following form:

  σ: $< \pi = \{l_0, m_0\}, y = 1 > \rightarrow \ldots < \pi = \{l_2, m_2\}, y = 1 > \rightarrow^{m_2}$
  $< \pi = \{l_2, m_3\}, y = 0 > \rightarrow^{m_3} < \pi = \{l_2, m_4\}, y = 0 > \rightarrow^{m_4}$
  $< \pi = \{l_2, m_0\}, y = 1 > \rightarrow^{m_0} < \pi = \{l_2, m_1\}, y = 1 > \rightarrow^{m_1}$
  $< \pi = \{l_2, m_2\}, y = 1 > \rightarrow \ldots$

  Such a computation visits infinitely many times the state $< \pi = \{l_2, m_2\}, y = 1 >$ where the transition l_2 is enabled, which implies that the requirement of compassion is violated.

  This allows us to conclude that the program satisfies the property of accessibility,

The mutual exclusion problem: the SPL program MUX_WHEN

local y: integer where y=1

```
P1:: [ ℓ0: loop forever do                      ]    P2:: [ m0: loop forever do                      ]
     [      [ ℓ1: noncritical              ]    ]  ||     [      [ m1: noncritical              ]    ]
     [      [ ℓ2: <when y>0 do y:=y-1>     ]    ]         [      [ m2: <when y>0 do y:=y-1>     ]    ]
     [      [ ℓ3: critical                 ]    ]         [      [ m3: critical                 ]    ]
     [      [ ℓ4: y:=y+1                   ]    ]         [      [ m4: y:=y+1                   ]    ]
     [ ℓ5:                                      ]         [ m5:                                      ]
```

The properties of exclusivity and accessibility

- It **satisfies** the property of **mutual exclusion**: it can be shown by means of an argument analogous to the one used for the previous program (the use of a grouped instruction in $l2$ and $m2$ is fundamental).
- It **does not satisfy** the property of **accessibility**: the previously described computation where $l2$ is enabled infinitely many times (but not constantly enabled) and never taken from a given point onwords is admissible (for $l2$ and $m2$ justice, and not compassion, is required).

The property of "communal" accessibility

- The proposed variant **satisfies** a weaker form of accessibility, called **communal accessibility**.

  Communal accessibility states that each state of a computation where a process reaches the end of its non-critical region, e.g., location $l2$ for $P1$, is followed by a state where some process, not necessarily the same, is within its critical region, that is, location $m3$ for $P2$ or location $l3$ for $P1$.

Producer-Consumer with a Bounded Buffer

```
local send, ack: channel [1..] of integer
        where send=Λ , ack=[1,...,1]

Prod:: [ local x, t: integer
         ℓ0: loop forever do
             [ ℓ1: produce x
               ℓ2: ack ⇒ t
               ℓ3: send ⇐ x ] ]
||
Cons:: [ local y: integer
         m0: loop forever do
             [ m1: send ⇒ y
               m2: ack ⇐ 1
               m3: consume y ] ]
```

where `ack` initially contains n occurrences of 1; $l2$ can be executed only if `ack` is not empty, $m1$ only if `send` is not empty.

The SPL program Fair-Merge

out b : channel of integer
local a1, a2 : channel of integer

```
P1:: [ local x1: integer                       ]  ||  P2:: [ local x2: integer                ]  ||
     [ ℓ0: loop forever do [ ℓ1: produce x1     ] ]      [ k0: loop forever do [ k1: produce x2 ] ]
     [                     [ ℓ2: a1 ⇐ 2•x1 + 1  ] ]      [                     [ k2: a2 ⇐ 2•x2  ] ]

Merger:: [ local y: integer                    ]  ||  Cons:: [ local z: integer                  ]
         [ m0: loop forever do                 ]             [ n0: loop forever do               ]
         [          [ m1a: a1 ⇒ y ]            ]             [        [ n1: b ⇒ z         ]      ]
         [      m1: [ or          ]            ]             [        [ n2: consume z     ]      ]
         [          [ m1b: a2 ⇒ y ]            ]
         [      m2: b ⇐ y                      ]
```