Università degli Studi di Udine

Master degree in Artificial Intelligence and Cybersecurity

Fair Transition Systems - Part 2

Angelo Montanari

SPL PROGRAMS

The structure of **SPL programs**:

- Syntax:
  P :: [declaration; [P_1::[l_1 : S_1; $\widehat{l_1}$:] || ... ||
  P_k::[l_k : S_k; $\widehat{l_k}$:]]]
- The names of the program (P) and of the processes (P_i) are optional.
- The syntax is similar to that of cooperation instructions, but there are not the *entry step* and the *exit step*.

The declaration

- The **declaration** consists of a list of instructions of the following form:
  `mode` $var_1, \ldots, var_n$: *type* `where` φ
  where `mode` specifies the mode/type of the declaration.

  It may assume the following values:
  - `in` (it specifies the input variables);
  - `local` (it specifies the local variables);
  - `out` (it specifies the output variables).

  The distinction between `local` and `out` variables is, in fact, a distinction of convenience, whose only goal is to improve program readability.

Programs and variables

- The program cannot modify the value of variables of type `in` (read-only variables); on the contrary, it can modify the value of `local` and `out` variables.
- The formula φ (optional) constrains the initial values of variables
  - `local`/`out`: it can only take the form $y = e$ (initialization), where the expression e can only include variables of type `in`;
  - `in`: there are not restrictions on the form of φ, that can be exploited to specify the set of admissible values for input variables.
- The logical conjunction of conditions is called **data pre-condition**.

Channel declaration

A special role is played by the **channel** declarations:

- **sinchronous** channels:
  `mode` $\alpha_1, \ldots, \alpha_n$: `channel of` *type*

- **asinchronous** channels:
  `mode` $\alpha_1, \ldots, \alpha_n$: `channel [1..] of` *type* `where` φ

  where `[1..]` denotes a buffer of unbounded capacity and φ specifies conditions on the initial content of the buffer (if φ is missing, the buffer is assumed to be initially empty).

An example of SPL program

calculation of the greatest common divisor with the Euclidean algorithm

```
in a, b: integer where a>0, b>0
local y1, y2: integer where y1=a, y2=b
out g: integer

[ l0: [ l1: while y1<>y2 do
          l2: [ l3: [ l9: await y1>y2; l4: y1:=y1-y2 ]
                or
                l5: [ l10: await y2>y1; l6: y2:=y2-y1 ] ]
      l7: g:=y1
  l8: ]
```

Labels

- **Labels** are used in SPL in a twofold way:
  - to univocally identify instructions;
  - to localize the control.
- More than one label can identify the same location of the control.

  The following **equivalence relation** on the set of labels allows one to identify labels associated with the same location:
  - from l : `[`$l_1 : S_1, \ldots, l_k : S_k$`]`, we have that $l \sim_L l_1$;
  - from l : `[`$l_1 : S_1$ `or ... or` $l_k : S_k$`]`, we have that $l \sim_L l_1 \sim_L \ldots \sim_L l_k$;
  - from l : `[local declaration;` $l_1 : S_1$`]`, we have that $l \sim_L l_1$.

Control locations

Control locations and classes of the equivalence relation $\sim_L$.

- A **control location** is an equivalence class $[l_i]$ of $\sim_L$.
- An example:
  - $[l_1] = \{l_0, l_1\}$;
  - $[l_4] = \{l_4\}$;
  - $[l_6] = \{l_6\}$;
  - $[l_7] = \{l_7\}$;
  - $[l_8] = \{l_8\}$;
  - $[l_2] = \{l_2, l_3, l_9, l_5, l_{10}\}$.

An example of SPL program (revisited)

calculation of the greatest common divisor with the Euclidean algorithm

```
in a, b: integer where a>0, b>0
local y1, y2: integer where y1=a, y2=b
out g: integer

[ ℓ1: while y1<>y2 do
        ℓ2: [ ℓ2a: await y1>y2; ℓ4: y1:=y1-y2
            or
              ℓ2b: await y2>y1; ℓ6: y2:=y2-y1 ]
  ℓ7: g:=y1
  ℓ8: ]
```

Post-Locations - 1

At the end of the execution of any instruction, with the exception of the body of the program, a unique location is reached, called the **post-location** of the instruction.

The connection between the post-location of a compound instruction and the post-locations of its component instructions is expressed by the following rules:

- Concatenation: $S = [l_1{:}S_1;\ \ldots;\ l_k{:}S_k]$
  - $post(S) = post(S_k)$
  - for all i, with $i < k$, $post(S_i) = l_{i+1}$

Post-Locations - 2

- Cooperation: $S = [l_1{:}S_1; \widehat{l_1}{:}] \,||\, \ldots \,||\, [l_k{:}S_k; \widehat{l_k}{:}]$
  - $post(S_i) = [\widehat{l_i}]$
  - for all i, j, con $i \neq j$, $post(S_i) \neq post(S_j)$ (there is the exit step)
- Selection: $S = [l_1{:}S_1$ or $\ldots$ or $l_k{:}S_k]$
  - $post(S) = post(S_1) = \ldots = post(S_k)$
- Condizional: $S = \mathit{if}\ c\ \mathit{then}\ S_1\ \mathit{else}\ S_2$
  - $post(S) = post(S_1) = post(S_2)$
- While: $S = [l : \mathit{while}\ c\ \mathit{do}\ S']$
  - $post(S') = [l]$
- Block: $S = [\mathit{local\ declaration},\ S']$
  - $post(S) = post(S')$

An example

Let us consider the above-described SPL program for the calculation of the greatest common divisor.

The post-locations are the following:

- $post(l_1) = [l_7]$
- $post(l_2) = post(l_4) = post(l_6) = [l_1]$
- $post(l_{2a}) = [l_4]$
- $post(l_{2b}) = [l_6]$
- $post(l_7) = [l_8]$

Remark: we identified instructions with their labels.

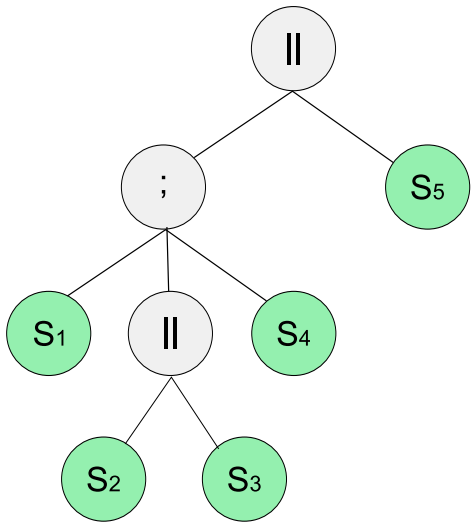
The ancestor relation on the set of instructions of an SPL program - 1

- If S' is a sub-instruction of S, S is called an **ancestor** of S'.
  - For all S, S is an *ancestor* of S (reflexivity);
  - if S is an *ancestor* of S_1 and S_1 is an *ancestor* of S_2, then S is an *ancestor* of S_2 (transitivity).
- S is called a *common ancestor* of S_1 and S_2 if it is an *ancestor* of both S_1 and S_2.
- S is the *least common ancestor (LCA)* of S_1 and S_2 if
  - S is a *common ancestor* of S_1 and S_2;
  - any *common ancestor* of S_1 and S_2 is also an *ancestor* of S.

The ancestor relation on the set of instructions of an SPL program - 2

An **example**: [S_1; [S_2 || S_3]; S_4] || S_5.

- LCA of S_2 and S_3 is $S_2||S_3$;
- LCA of S_2 and $S_2||S_3$ is $S_2||S_3$;
- LCA of S_2 and S_4 is $[S_1;[S_2||S_3];S_4]$;
- LCA of S_2 and S_5 is the whole instruction.



Parallel instructions and instructions in conflict

- If S_i and S_j have as their LCA a cooperation instruction S, with $S_i \neq S_j$, $S_i \neq S$, and $S_j \neq S$, S_i and S_j are called **parallel instructions**;
- otherwise, they are called **instructions in conflict**.

- In the previous example,
  - S_2 and S_3 are parallel instructions;
  - S_2 and $S_2||S_3$ are instructions in conflict;
  - S_2 and S_4 are instructions in conflict;
  - S_2 and S_5 are parallel instructions.

Semantics and computations of SPL programs

- The **semantics** of an SPL program is an FTS

$$\langle V, \theta, \mathcal{T}, \mathcal{J}, \mathcal{C} \rangle$$

- The **computations** of an SPL program are the computations of the corresponding FTS.

V: THE VARIABLES - 1

- V collects the set $Y = \{y_1, y_2, ..., y_n\}$ of **program variables** and a single **control variable** π that takes as values the locations of the control during the execution of the program:

$$V = \{y_1, y_2, ..., y_n\} \cup \{\pi\}$$

- **An example**: the algorithm to compute the greatest common divisor.
  - $V = \{\pi, a, b, y_1, y_2, g\}$, where the domain of π is the powerset of $\{[l_1], [l_2], [l_4], [l_6], [l_7], [l_8]\}$.

V: THE VARIABLES - 2

Observations

- the value of the variable π is a subset of $\{[l_1], ..., [l_k]\}$ (limit case: a singleton);
- for any asynchronous channel α, a variable α is introduced, with $\alpha \in Y$, whose values are lists of elements belonging to the type of the channel;
- no variable is introduced for synchronous channels.

Useful **abbreviations**:

- for all labels l of the program, at_l is a shorthand for $[l] \in \pi$;
- for all labels l of the program, at'_l is a shorthand of $[l] \in \pi'$.

θ: THE INITIAL CONDITION

Given the program
P :: [declaration; [P_1::[$l_1 : S_1; \widehat{l_1}$:] || ... ||
P_k::[$l_k : S_k; \widehat{l_k}$:]]]

the **initial condition** θ consists of the conjunction φ of the formulas occurring in the `where` clauses of the variable declarations plus the condition on the control that forces the set of input locations $([l_1], ..., [l_k])$ to be the initial value of the control variable:

$$\theta = \varphi \wedge \pi = \{[l_1], ..., [l_k]\}$$

An example: the algorithm to compute the greatest common divisor.

$$\theta = a > 0 \wedge b > 0 \wedge y_1 = a \wedge y_2 = b \wedge \pi = \{[l_1]\}$$

$\mathcal{T}$: THE TRANSITIONS

The **set of transitions** $\mathcal{T}$

- The set $\mathcal{T}$ consists of all and only the transitions that implement the instructions of the given SPL program plus the idling transition τ_I (whose transition relation is $\rho_I : V' = V$).
- With the exception of τ_I, all transitions are self-disabling, that is, their execution changes at least the value of the control variable π.

$\mathcal{T}$: SOME USEFUL ABBREVIATIONS

- The expression $move(L, \widehat{L})$ is an shorthand for $L \subseteq \pi \wedge \pi' = (\pi - L) \cup \widehat{L}$
- As a special case, we write a $move(l, \widehat{l})$ for $move(\{[\![l]\!]\}, \{[\![\widehat{l}]\!]\})$ $(\equiv \{[\![l]\!]\} \subseteq \pi \wedge \pi' = (\pi - \{[\![l]\!]\}) \cup \{[\![\widehat{l}]\!]\})$.
- Since each transition does not change the values of most of the variables, we use the shorthand

$$pres(U) \equiv \bigwedge_{u \in U} (u' = u)$$

$\mathcal{T}$: BASIC INSTRUCTIONS - 1

- l: `skip`; $\widehat{l}$:
  - ρ_l: $move(l, \widehat{l}) \wedge pres(Y)$
  - it changes the location only.
- l: $\overline{u}:=\overline{e}$; $\widehat{l}$:
  - ρ_l: $move(l, \widehat{l}) \wedge \overline{u}' = \overline{e} \wedge pres(Y - \{\overline{u}\})$
  - it changes the location and the values of the variables in $\overline{u}$.
- l: `await` c; $\widehat{l}$:
  - ρ_l: $move(l, \widehat{l}) \wedge c \wedge pres(Y)$
  - the transition is enabled when the condition c is true; when it is executed, it changes the location only.

$\mathcal{T}$: BASIC INSTRUCTIONS - 2

- Asynchronous send: l: $\alpha \Leftarrow e$; $\widehat{l}$:
  - ρ_l: $move(l, \widehat{l}) \wedge \alpha' = \alpha \cdot e \wedge pres(Y - \{\alpha\})$, where $\cdot$ is the concatenation operation;
  - the new value of α is given by the concatenation of the current value of α with the value of e.

- Asynchronous receive: l: $\alpha \Rightarrow u$; $\widehat{l}$:
  - ρ_l: $move(l, \widehat{l}) \wedge |\alpha| > 0 \wedge \alpha = u' \cdot \alpha' \wedge pres(Y - \{u, \alpha\})$
  - the transition is enabled only if the channel is not empty.

$\mathcal{T}$: BASIC INSTRUCTIONS - 3

- Synchronous send/receive:
  l: $\alpha \Leftarrow e$; $\widehat{l}$; $\quad m$: $\alpha \Rightarrow u$; $\widehat{m}$:
  - $\rho_{l,m}$: $move(\{l, m\}, \{\widehat{l}, \widehat{m}\}) \wedge u' = e \wedge pres(Y - \{u\})$
  - the transition is enabled if both at_l and at_m are satisfied.

- Request: l: `request` r; $\widehat{l}$:
  - ρ_l: $move(l, \widehat{l}) \wedge r > 0 \wedge r' = r - 1 \wedge pres(Y - \{r\})$
  - the transition is enabled only if r is greater than 0; when it is executed, it decrements the value of r by 1.

- Release: l: `release` r; $\widehat{l}$:
  - ρ_l: $move(l, \widehat{l}) \wedge r' = r + 1 \wedge pres(Y - \{r\})$
  - when it is executed, it increments the value of r by 1.

$\mathcal{T}$: SCHEMATIC INSTRUCTIONS - 1

- l: `noncritical`; $\widehat{l}$:
  - ρ_l: $move(l, \widehat{l}) \wedge pres(Y)$
- l: `critical`; $\widehat{l}$:
  - ρ_l: $move(l, \widehat{l}) \wedge pres(Y)$

As we will see when we will describe the components of the FTS that encode the fairness conditions, the second transition differs from the first one in the termination condition.

$\mathcal{T}$: SCHEMATIC INSTRUCTIONS - 2

- l: `produce` x; $\widehat{l}$:
  - ρ_l: $move(l, \widehat{l}) \wedge x' \neq 0 \wedge pres(Y - \{x\})$
- l: `consume` y; $\widehat{l}$:
  - ρ_l: $move(l, \widehat{l}) \wedge pres(Y)$

Termination is required for both transitions.

$\mathcal{T}$: COMPOUND INSTRUCTIONS - 1

- l: `if` c `then` $l_1 : S_1$ `else` $l_2 : S_2$; $\widehat{l}$:
  - ρ_l: $\rho_l^T \vee \rho_l^F$, where
    ρ_l^T: $move(l, l_1) \wedge c \wedge pres(Y)$
    ρ_l^F: $move(l, l_2) \wedge \neg c \wedge pres(Y)$
  - it generates a unique transition, not a pair of them, and thus it is always enabled (one of the two disjuncts is satisfied).
  - a special case:
    l: `if` c `then` $l_1 : S_1$; $\widehat{l}$
    the disjunct ρ_l^F is modified in the following way:
    ρ_l^F: $move(l, \widehat{l}) \wedge \neg c \wedge pres(Y)$

$\mathcal{T}$: COMPOUND INSTRUCTIONS - 2

- l: `while` c `do` $[\widetilde{l} : \widetilde{S}]; \widehat{l}$:
  - ρ_l: $\rho_l^T \vee \rho_l^F$, where ρ_l^T: $move(l, \widetilde{l}) \wedge c \wedge pres(Y)$ e ρ_l^F: $move(l, \widehat{l}) \wedge \neg c \wedge pres(Y)$
  - it generates a unique transition, not a pair of them, and thus it is always enabled (one of the two disjuncts is satisfied);
  - the semantics of the sub-instruction $\widetilde{S}$ and its labels: $\widetilde{l} : \widetilde{S}\ l$:
- l: $[\ [l_1 : S_1; \widehat{l_1} :]\ \ ||\ \ \ldots\ \ ||\ \ [l_k : S_k; \widehat{l_k} :]]; \widehat{l}$:
  - ρ_l^E: $move(l, \{l_1, ..., l_k\}) \wedge pres(Y)$
  - ρ_l^X: $move(\{\widehat{l_1}, ..., \widehat{l_k}\}, \widehat{l}) \wedge pres(Y)$
  - the case of the body of the program is excluded.

Each transition belonging to the implementation of the compound instructions of concatenation, selection, and block can be associated with one of their sub-instructions.

$\mathcal{T}$: GROUPED (COMPOSITE) INSTRUCTIONS - 1

The **grouped (composite) instructions**: $\langle S \rangle$

As the grouped instructions are atomic instructions, which are executed in one single step, we must determine their global effect.

To this end, we introduce a **relation of data transformation** $\delta[S]$ that only refers to the variables in Y.

$\mathcal{T}$: GROUPED (COMPOSITE) INSTRUCTIONS - 2

In the **simplest cases** (`skip`, assignment, `await`, asynchronous `send`, asynchronous `receive`), $\delta[S]$ is defined as ρ_l, with l label of S, devoid of $move(l, \widehat{l})$.

In the **most complex cases**, it is defined as follows:

- when:
  - $\delta[\texttt{when } c \texttt{ do } S]$: $c \wedge \delta[S]$
- conditional:
  - $\delta[\texttt{if } c \texttt{ then } S_1 \texttt{ else } S_2]$: $(c \wedge \delta[S_1]) \vee (\neg c \wedge \delta[S_2])$
  - a special case:
    $\delta[\texttt{if } c \texttt{ then } S]$: $(c \wedge \delta[S]) \vee (\neg c \wedge pres(Y))$

$\mathcal{T}$: GROUPED (COMPOSITE) INSTRUCTIONS - 3

- selection:
  - $\delta[S_1 \texttt{ or } \ldots \texttt{ or } S_k]$: $\delta[S_1] \vee \ldots \vee \delta[S_k]$
- concatenation:
  - $\delta[S_1; S_2]$: $\exists \widetilde{Y}(\delta[S_1](Y, \widetilde{Y}) \wedge \delta[S_2](\widetilde{Y}, Y'))$

An example

$\delta[x := x + 2; x := x - 3]$:

$\exists \widetilde{x}(\widetilde{x} = x + 2 \wedge x' = \widetilde{x} - 3) \wedge pres(Y - \{x\})$

$\mathcal{T}$: GROUPED (COMPOSITE) INSTRUCTIONS - 4

The **grouped instructions**

- $l\colon \langle S \rangle; \widehat{l}\,\colon$
  - $\rho_l\colon\ \delta[S] \wedge \mathit{move}(l, \widehat{l})$ (it obviously differs from the semantics of $l\colon\ S; \widehat{l}\colon$).

Grouped instructions that include **synchronous communications**.

- $l\colon \langle T_1;\ \alpha \Leftarrow e;\ S_1 \rangle; \widehat{l}\colon$
  $m\colon \langle T_2;\ \alpha \Rightarrow u;\ S_2 \rangle; \widehat{m}\colon$
  - For simplicity, let us assume variables in $T_1 \cup S_1$ to be disjoint from those in $T_2 \cup \{u\} \cup S_2$ (the only common variable is the synchronous channel α). In such a case,

    $\rho_{l,m}\colon\ \mathit{move}(\{l, m\}, \{\widehat{l}, \widehat{m}\}) \wedge \delta[T_1; T_2; u := e; S_1; S_2]$

$\mathcal{J}$ AND $\mathcal{C}$: FAIRNESS CONDITIONS

Transitions for which we require **justice**:

- $\mathcal{J}$ includes all the transitions, with the exception of the idling transition and of the transitions associated with the occurrences of the instruction `noncritical` (control may remain indefinitely in front of them).

Transitions for which we require **compassion**:

- $\mathcal{C}$ includes all transitions associated with
  - the communication instructions `send` e `receive`;
  - grouped instructions that include communication instructions;
  - the instruction `request` for the management of semaphores (but not the instruction `release`, that releases a shared resource – an operation for which there is no competition).