Università degli Studi di Udine

Master degree in Artificial Intelligence and Cybersecurity

Fair Transition Systems - Part 1

Angelo Montanari

Basic Notions - 1

Fair Transition Systems (FTSs) as a way to model reactive programs/systems.

To start with, we ask ourselves: how can we represent FTSs?

- The state of an FTS is expressed by means of (the values of) a given set of variables belonging to a variable universe $\mathcal{V}$ (*vocabulary*).
- Variables in $\mathcal{V}$ *are typed (sorted)*.
- For every variable $x \in \mathcal{V}$, we assume that there exists a "variable" $x' \in \mathcal{V}$ (if variable x is associated with a given state, variable x' is associated with the next state).

Basic notions - 2

- An FTS can be described by means of **first-order logic**:
  - terms/expressions are built on variables in $\mathcal{V}$ and constants by (possibly) applying the usual functions;
  - atomic formulas are either proposition letters (Boolean variables) or predicates applied to terms;
  - **state formulas** (or assertions) are obtained from atomic ones by applying Boolean connectives and quantifiers.
- A **state** is an interpretation of $\mathcal{V}$ (that satisfies the type constraint on variables), that assigns to each variable $u \in \mathcal{V}$ a value $s[u]$ belonging to the domain of u.

  We denote by Σ the set of all the states (if such a set is infinite, we have an infinite state system).

Fair Transition Systems

A *Fair Transition System* (FTS) is a tuple

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

where

- $V = \{u_1, ..., u_n\}$ is a set of **system variables**.
  V can be partitioned into:
  - the *control variable*, and
  - the *data variables*.
- θ is a satisfiable state formula that characterizes the possible initial states of the system (**initial condition**). A state that satisfies θ is called an initial state.

The finite set of transitions

- $\mathcal{T}$ is a **finite set** of **transitions**
  - each transition $\tau \in \mathcal{T}$ is a function
    $$\tau : \Sigma \to 2^{\Sigma}$$
    that maps each state $s \in \Sigma$ into a set of states $\tau(s) \subseteq \Sigma$;
  - each state in $\tau(s)$ is called a τ-*successor* of s;
  - τ is *enabled* in s if $\tau(s) \neq \emptyset$, otherwise τ is *disable*.

Remark. The triple $\langle V, \theta, \mathcal{T} \rangle$ defines a classic transition system.

Weak and strong fairness conditions

- $\mathcal{J} \subseteq \mathcal{T}$ is the set of transitions for which we require **justice** (**weak fairness**), that is, we exclude the existence of computations where they are constantly enabled from a given state on and never taken.
- $\mathcal{C} \subseteq \mathcal{T}$ is the set of transitions for which we require **compassion** (**strong fairness**), that is, we exclude the existence of computations where they are enabled infinitely many times and taken a finite number of times only.

A couple of remarks

Remark 1. Strong fairness implies weak fairness, but not vice versa.

Remark 2. It is not possible to "finitely" check, that is, by analysing a finite prefix of it, whether or not a given computation satisfies the fairness conditions

How do we represent transitions?

Each transition $\tau \in \mathcal{T}$ can be represented by a formula of first-order logic $\rho_\tau(V, V')$, called **transition relation**.

The transition relation $\rho_\tau(V, V')$ expresses a relationship between a state s and each of its τ-successors $s' \in \tau(s)$ (by $x, y, z, \ldots$ we refer to the values of variables in s, while by $x', y', z', \ldots$ we refer to their values in s').

Example: formula $x' = x + 1$ states that the value of x at s' ($s'[x]$) is equal to the value of x at s plus 1 ($s[x] + 1$).

The state s' is a successor of the state s via transition τ (s' is a τ-successor of s) if $\rho_\tau(V, V')$ evaluates to true if we interpret each $x \in V$ as $s[x]$ and each $x' \in V'$ as $s'[x]$.

Enabling condition

We can express the **enabling condition** of a transition τ, denoted by $En(\tau)$, at a state s by means of the formula $\exists V' \rho_\tau(V, V')$.

The set of transitions $\mathcal{T}$ always includes the **idling transition**, denoted by τ_I, which is defined as follows: $\rho_I : V' = V$, that is, s' is a τ_I-successor of s if and only if s and s' agree on the value of all system variables, including the control variable.

Neither justice nor compassion is requested for τ_I: it may happen that τ_I is always enabled (as it is), and never taken.

τ_I can be exploited to "complete" finite computations.

All transitions $\tau \in \mathcal{T}$ different from τ_I are called **diligent transitions**.

Computations of an FTS - 1

Let us consider an infinite sequence of states $\sigma = s_0, s_1, s_2, \ldots$, where $s_i \in \Sigma$ for all i.

We say that $\tau \in \mathcal{T}$ is **enabled** at position k of the sequence if τ is enabled at state s_k.

We say that $\tau \in \mathcal{T}$ is **executed** (taken) at position k of the sequence if $s_{k+1} \in \tau(s_k)$.

It is worth pointing out that more than one transition can be (considered as) taken at the same position.

Computations of an FTS - 2

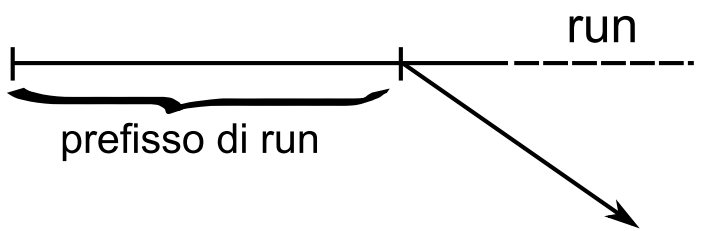
A sequence σ is a **computation of an FST** P (**P-computation**) if it satisfies the following conditions:

- (**initiality**) s_0 satisfies θ (s_0 is an initial state);
- (**consequentiality**) for all $j = 0, 1, 2, ...$, s_{j+1} is a τ-successor of s_j for some τ;
- (**justice**) for all $\tau \in \mathcal{J}$, it is never the case that τ is constantly enabled from a given position k on and never taken;
- (**compassion**) for all $\tau \in \mathcal{C}$, it is never the case that, from a given position k on, τ is enabled an infinite number of times, and taken only finitely many times.

Justice and compassion conditions are called **fairness requirements**.

Properties of Computations

- A state s is P-**accessible** if it occurs in some P-computation.
- Each state which is the τ-successor of a P-accessible state, for some $\tau \in \mathcal{T}$, is P-accessible.
- Every sequence σ that satisfies the initiality and consequentiality requirements, but not necessarily fairness ones, is called a **run** of P.
  Every finite prefix of a run is a prefix of a computation as well (each state s that occurs in a run is thus P-accessible).



SPL: a simple programming language

Let us introduce now a simple **programming language**, called **SPL**, that allows one to synthesize reactive programs.

It features a good compromise between simplicity and expressiveness.

- An SPL program is a finite sequence of (possibly labeled) instructions.
- The set of **SPL instructions** can be partitioned into four main classes:
  - basic instructions;
  - schematic instructions;
  - compound instructions;
  - grouped composite instructions.

Basic instructions

The set of **basic instructions**:

- skip;
- assignment;
- await;
- send/receive (synchronous and asynchronous communications);
- request/release (primitives for the management of semaphores).

Skip

The **skip** instruction.

- Syntax: `skip`
- Semantics: it has no effect, apart from moving the control from before to after the instruction.

Assignment

The instruction of (multiple) **assignment**.

- Syntax: $(u_1, \ldots, u_k)$:= $(e_1, \ldots, e_k)$
  - $u_1, \ldots, u_k$ is a sequence of variables;
  - $e_1, \ldots, e_k$ is a sequence of expressions/terms.
- Semantics: it simultaneously executes the assignments $u_1 := e_1, \ldots, u_k := e_k$.
- Particular cases:
  - vectors/sequences (compact notation): $\overline{u}$:= $\overline{e}$;
  - singletons: u := e.

Await

The **await** instruction.

- Syntax: `await` c
  - c is a Boolean expression (called *guard*).
- Semantics: it waits for the guard to come true. Once it happens, it transfers the control from before to after the instruction.
- It does not change the value of any data variable.
- A particular case:
  - `await false` $\equiv$ `halt`
    (it stops the computation).

Istructions for message passing: Send

The **send** instruction.

- Syntax: $\alpha \Leftarrow e$
  - α is a channel (a channel is a system variable of special kind);
  - e is an expression whose type is compatible to the one of α.
- Semantics: e is evaluated and its value is assigned to the channel α.

Istructions for message passing: Receive

The **receive** instruction.

- Syntax: $\alpha \Rightarrow u$
  - α is a channel;
  - u is a variable whose type is compatible to the one of α.
- Semantics: the value of α is assigned to the variable u.
- If the channel does not contain any value, the execution of the instruction is suspended/delayed (as we will see, this may happen with asynchronous channels only).

Instructions for the management of semaphores: Request/Release

The instructions **request** e **release**.

- Syntax: `request` r
- Semantics: the instruction is executed only if $r > 0$ (otherwise, its execution is suspended/delayed), and it decrements the value of r by 1.

- Syntax: `release` r
- Semantics: the instruction increments the value of r by 1.

Schematic instructions

Schematic instructions represent sequences of instructions whose details are not of interest (only termination conditions matter).

Schematic instructions are particularly useful in the modeling of some classical problems:

- mutual exclusion: noncritical/critical;
- producer/consumer paradigm: produce/consume.

Mutual exclusion: noncritical/critical

The **noncritical** instruction.

- Syntax: `noncritical`
- Semantics: it condenses in a unique instruction an activiy of the program which is **not critical** from the point of view of the management of mutual exclusion (use of private resources).
  Termination is not required (it is equivalent to τ_l).

The **critical** instruction.

- Syntax: `critical`
- Semantics: it condenses in a unique instruction an activiy of the program which is **critical** from the point of view of the management of mutual exclusion (use of shared resources), that requires some coordination among processes.
  It must terminate (some fairness conditions must be imposed).

The producer/consumer paradigm: produce/consume

The **produce** instruction.

- Syntax: `produce` x, where x is a variable.
- Semantics: it assigns to x a value different from 0 (it is the only schematic instruction that changes the value of data variables). **It must terminate**.

The **consume** instruction.

- Syntax: `consume` y, where y is a variable.
- Semantics: it models resource consumption in the producer/consumer schema. Let us assume that its execution does not change the value of y. **It must terminate**.

Compound instructions

The set of **compound instructions** includes the following instructions:

- conditional instruction;
- concatenation instruction;
- selection instruction;
- while instruction;
- cooperation instruction;
- block instruction.

The conditional instruction

The **conditional** instruction.

- Syntax: `if` c `then` S_1 `else` S_2
  - c is a Boolean condition;
  - S_1 are S_2 instructions.
- Semantics: c is evaluated; if c is true, then S_1 is executed; otherwise, S_2 is executed.
- Unlike `await`, it is always enabled.
- A particular case:
  - `if` c `then` S_1 (if c is false, the execution of the instruction terminates in one step).

The concatenation instruction

The **concatenation** instruction.

- Syntax: $S_1\,;\ S_2; \ldots ; S_k$
  - each S_i, with $i \in \{1, \ldots, k\}$, is an instruction.
- Semantics: the first step of the execution of the instruction coincides with the first step of the execution of S_1.
- A particular case:
  - `when` c `do` $S \equiv$ `await` c`;` S (guarded command).

The selection instruction

The **selection** instruction.

- Syntax: S_1 `or` S_2 `or` ... `or` S_k
  - S_i, with $i \in \{1, \ldots, k\}$, are instructions.
- Semantics: it selects the subset of enabled instructions and nondeterministically chooses one of them as the instruction to execute.
- If no instruction is enabled, the execution of the selection instruction is suspended/delayed.
- This instruction is often used in combination with the `when` instruction:
  `if` $c_1 \rightarrow s_1 \; \square \; \ldots \; \square \; c_k \rightarrow s_k$ `fi` $\equiv$
  `[when` c_1 `do` s_1`]` `or` ... `or` `[when` c_k `do` s_k`]`

The while instruction

The **while** instruction.

- Syntax: `while` c `do` S
  - c is a Boolean expression;
  - S is an instruction.
- Semantics: c is evaluated; if c is true, then S is executed; otherwise, the execution of the instruction terminates.
- The execution of this instruction is never suspended.
- Special cases:
  - `while` $TRUE$ `do` $S \equiv$ `loop forever` S
  - `for` $i := 1$ `to` n `do` $S \equiv$
    $i := 1$; `while` $i \leq n$ `do` $[S;\ i := i + 1]$

The cooperation instruction - 1

The **cooperation** instruction.

- It aims at modeling the parallel execution of a given set of instructions.
- Syntax: $l:\quad [l_1 : S_1; \widehat{l_1} :] \;||\; \ldots \;||\; [l_k : S_k; \widehat{l_k} :]\,;\; \widehat{l}:$
  - l, $\widehat{l}$, l_i, and $\widehat{l_i}$, with $i \in \{1, \ldots, k\}$, are **labels**;
  - S_i, with $i \in \{1, \ldots, k\}$, are instructions.

The cooperation instruction - 2

- Semantics:
  - the first step of the execution of the cooperation instruction is called **entry step**; it transfers the control from l to $l_1, \ldots, l_k$ (it starts the parallel execution of instructions $S_1, \ldots, S_k$);
  - the last step of the execution of the cooperation instruction is called **exit step**; it transfers the control from $\widehat{l_1}, \ldots, \widehat{l_k}$ to $\widehat{l}$ (to end the execution of the cooperation instruction, the execution of all the component instructions must be completed).

The block instruction

The **block** instruction.

- Syntax: `[ local declaration;` S `]`
  - `local declaration`: it is a list of instructions of variable declaration of the form:
    `local` $var_1, \ldots, var_n$`:` $type$ `where` φ
    the formula φ (optional) is of the form $y_1 = e_1, \ldots, y_m = e_m$, where $y_1, \ldots, y_m$ are variables and $e_1, \ldots, e_m$ are expressions, that comply with the types of the variables, whose values depend on variables that have been declared outside the block (and on constants);
  - S is an instruction, called *body* of the block.
- The instructions of declaration are used to initialize the value of local variables.

Grouped composite instructions - 1

- They are used to make some (not all) compound instructions atomic (in analogy to what is done in databases by means of transactions).
- The definition of the notion of grouped composite instructions is built on that of composite instruction.
- A **composite instruction** is defined as follows:
  - `skip`, assignement, `await`, `send`, and `receive` are basic composite instructions;
  - if $S, S_1, \ldots, S_k$ are composite instructions, the following instructions are composite instructions as well:
    `when` c `do` S;
    `if` c `then` S_1 `else` S_2;
    `[`S_1 `or` ... `or` S_k`]`;
    `[`S_1;...;S_k`]`.

Grouped composite instructions - 2

- If S is a composite instruction, $\langle S \rangle$ is a **grouped composite instruction** provided that:
  - a grouped composite instruction that includes a synchronous communication instruction (`send`/`receive`) is of the form $\langle S_1;\ C;\ S_2 \rangle$, where C is a synchronous communication instruction and S_1 and S_2 do not include any other communication instruction (S_1 ed S_2 may be missing), that is, grouped composite instructions may include at most one synchronous `send` or `receive` instruction;
  - a grouped composite instruction cannot include two or more asynchronous communication instructions (`send`/`receive`) operating on the same channel.

An example. Let α e β be two asynchronous channels:

- $\langle \ldots \alpha \Leftarrow 1 \ldots \alpha \Leftarrow 3 \ldots \rangle$ NO;
- $\langle \ldots \alpha \Leftarrow 1 \ldots \beta \Leftarrow 3 \ldots \rangle$ YES.