Classification of Programs

There are two classes of programs:

**Computational Programs:** Run in order to produce a final result on termination.

Can be modeled as a black box.

\[ x \rightarrow y \]

Specified in terms of Input/Output relations.

**Example:**

The program which computes

\[ y = 1 + 3 + \cdots + (2x - 1) \]

Can be specified by the requirement

\[ y = x^2. \]
Reactive Programs

Programs whose role is to maintain an ongoing interaction with their environments, rather than produce a final result upon termination.

Examples: Air traffic control system, Programs controlling mechanical devices such as a train, a plane, or ongoing processes such as a nuclear reactor.

Termination is not necessarily expected, and the important functionality is interaction with the environment.

Can be viewed as a green cactus (?)

Such programs must be specified and verified in terms of their behaviors.
A Framework for Reactive Systems Verification

- A computational model providing an abstract syntactic base for all reactive systems. We use fair Discrete systems (FDS).
- A Specification Language for specifying systems and their properties. We use linear temporal logic (LTL).
- An Implementation Language for describing proposed implementations (both software and hardware). We use SPL, a simple programming language.
- Verification Techniques for validating that an implementation satisfies a specification. Practiced approaches:
  - A deductive methodology based on theorem-proving methods. Can accommodate infinite-state systems, but requires user interaction.


**Fair Discrete Systems**

A fair discrete system (FDS) $\mathcal{D} = \langle V, \mathcal{O}, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ consists of:

- **$V$** – A finite set of typed state variables. A $V$-state $s$ is an interpretation of $V$. $\Sigma_V$ – the set of all $V$-states.
- **$\mathcal{O} \subseteq V$** – A set of observable variables.
- **$\Theta$** – An initial condition. A satisfiable assertion that characterizes the initial states.
- **$\rho$** – A transition relation. An assertion $\rho(V, V')$, referring to both unprimed (current) and primed (next) versions of the state variables. For example, $x' = x + 1$ corresponds to the assignment $x := x + 1$.
- **$\mathcal{J} = \{J_1, \ldots, J_k\}$** A set of justice (weak fairness) requirements. Ensure that a computation has infinitely many $J_i$-states for each $J_i$, $i = 1, \ldots, k$.
- **$\mathcal{C} = \{\langle p_1, q_1 \rangle, \ldots \langle p_n, q_n \rangle \}$** A set of compassion (strong fairness) requirements. Infinitely many $p_i$-states imply infinitely many $q_i$-states.
A Simple Programming Language: SPL

A language allowing composition of parallel processes communicating by shared variables as well as message passing.

Example: Program ANY-Y

Consider the program

\[
\begin{align*}
x, y : \text{natural initially } x &= y = 0 \\
\ell_0 : \text{ while } x = 0 \text{ do } \\
&\quad [\ell_1 : y := y + 1] \\
\ell_2 : \text{ } &\quad \| \\
\text{ } &\quad [m_0 : x := 1] \\
\text{ } &\quad [m_1 : ]
\end{align*}
\]

\[\begin{align*}
- P_1 - &\quad - P_2 -
\end{align*}\]
The Corresponding FDS

- State Variables $V$: \[
\begin{pmatrix}
x, y & : & \text{natural} \\
\pi_1 & : & \{\ell_0, \ell_1, \ell_2\} \\
\pi_2 & : & \{m_0, m_1\}
\end{pmatrix}
\]
- Initial condition: $\Theta: \pi_1 = \ell_0 \land \pi_2 = m_0 \land x = y = 0$.
- Transition Relation: $\rho: \rho_I \lor \rho_{\ell_0} \lor \rho_{\ell_1} \lor \rho_{m_0}$, with appropriate disjunct for each statement. For example, the disjuncts $\rho_I$ and $\rho_{\ell_0}$ are

\[
\rho_I: \quad \pi'_1 = \pi_1 \land \pi'_2 = \pi_2 \land x' = x \land y' = y
\]

\[
\rho_{\ell_0}: \quad \pi_1 = \ell_0 \land \left( x = 0 \land \pi'_1 = \ell_1 \lor x \neq 0 \land \pi'_1 = \ell_2 \right) \land \pi'_2 = \pi_2 \land x' = x \land y' = y
\]

- Justice set: $\mathcal{J}: \{-\text{at}_0, -\text{at}_1, -\text{at}_m\}$.
- Compassion set: $\mathcal{C}: \emptyset$. 
Computations

Let $\mathcal{D}$ be an FDS for which the above components have been identified. The state $s'$ is defined to be a $\mathcal{D}$-successor of state $s$ if

$$\langle s, s' \rangle \models \rho_{\mathcal{D}}(V, V').$$

We define a computation of $\mathcal{D}$ to be an infinite sequence of states

$$\sigma : s_0, s_1, s_2, \ldots,$$

satisfying the following requirements:

- **Initiality:** $s_0$ is initial, i.e., $s_0 \models \Theta$.
- **Consecution:** For each $j = 0, 1, \ldots$, state $s_{j+1}$ is a $\mathcal{D}$-successor of state $s_j$.
- **Justice:** For each $J \in \mathcal{J}$, $\sigma$ contains infinitely many $J$-positions.
- **Compassion:** For each $\langle p, q \rangle \in \mathcal{C}$, if $\sigma$ contains infinitely many $p$-positions, it must also contain infinitely many $q$-positions.
Examples of Computations

Identification of the FDS $D_P$ corresponding to a program $P$ gives rise to a set of computations $\text{Comp}(P) = \text{Comp}(D_P)$.

The following computation of program $\text{ANY-Y}$ corresponds to the case that $m_0$ is the first executed statement:

$$\langle \pi_1: \ell_0, \pi_2: m_0; x: 0, y: 0 \rangle \xrightarrow{m_0} \langle \pi_1: \ell_0, \pi_2: m_1; x: 1, y: 0 \rangle \xrightarrow{\ell_0}$$

$$\langle \pi_1: \ell_2, \pi_2: m_1; x: 1, y: 0 \rangle \xrightarrow{\tau_I} \ldots \xrightarrow{\tau_I} \ldots$$

The following computation corresponds to the case that statement $\ell_1$ is executed before $m_0$.

$$\langle \pi_1: \ell_0, \pi_2: m_0; x: 0, y: 0 \rangle \xrightarrow{\ell_0} \langle \pi_1: \ell_1, \pi_2: m_0; x: 0, y: 0 \rangle \xrightarrow{\ell_1}$$

$$\langle \pi_1: \ell_0, \pi_2: m_0; x: 0, y: 1 \rangle \xrightarrow{m_0} \langle \pi_1: \ell_0, \pi_2: m_1; x: 1, y: 1 \rangle \xrightarrow{\ell_0}$$

$$\langle \pi_1: \ell_2, \pi_2: m_1; x: 1, y: 1 \rangle \xrightarrow{\tau_I} \ldots \xrightarrow{\tau_I} \ldots$$

In a similar way, we can construct for each $n \geq 0$ a computation that executes the body of statement $\ell_0$ $n$ times and then terminates in the final state

$$\langle \pi_1: \ell_2, \pi_2: m_1; x: 1, y: n \rangle.$$
A Non-Computation

While we can delay termination of the program for an arbitrary long time, we cannot postpone it forever.

Thus, the sequence

\[
\langle \pi_1 : \ell_0, \pi_2 : m_0 ; x : 0, y : 0 \rangle \xrightarrow{\ell_0} \langle \pi_1 : \ell_1, \pi_2 : m_0 ; x : 0, y : 0 \rangle \xrightarrow{\ell_1} \\
\langle \pi_1 : \ell_0, \pi_2 : m_0 ; x : 0, y : 1 \rangle \xrightarrow{\ell_0} \langle \pi_1 : \ell_1, \pi_2 : m_0 ; x : 0, y : 1 \rangle \xrightarrow{\ell_1} \\
\langle \pi_1 : \ell_0, \pi_2 : m_0 ; x : 0, y : 2 \rangle \xrightarrow{\ell_0} \langle \pi_1 : \ell_1, \pi_2 : m_0 ; x : 0, y : 2 \rangle \xrightarrow{\ell_1} \\
\langle \pi_1 : \ell_0, \pi_2 : m_0 ; x : 0, y : 3 \rangle \xrightarrow{\ell_0} \ldots
\]

in which statement \( m_0 \) is never executed is not an admissible computation. This is because it violates the justice requirement \( \neg at \_ m_0 \) contributed by statement \( m_0 \), by having no states in which this requirement holds.

This illustrates how the requirement of justice ensures that program ANY-Y always terminates.

Justice guarantees that every (enabled) process eventually progresses, in spite of the representation of concurrency by interleaving.
SPL: Syntax

Statements

- **skip** – A do-nothing statement.
- **$y := e$** – an assignment. Assign the value of expression $e$ to variable $y$.
- **await $b$** – Wait until the value of the boolean expression $b$ becomes true.
- **Compound Statements** – If $b$ is a boolean expression, and $S$, $S_1$, $S_2$ are statements, then so are
  - $S_1; S_2$ – Concatenation. Execute $S_1$ first and then $S_2$.
  - $[S]$ – Grouping.
  - **if $b$ then $S_1$ else $S_2$** – Conditional. Execute $S_1$ if $b$ evaluates to 1 (true). Otherwise, execute $S_2$.
  - **while $b$ do $S$** – a while statement. Repeatedly execute $S$ as long as $b$ evaluates to 1. If initially $b \sim 0$ then this is equivalent to **skip**.

- **Abbreviations**
  - **if $b$ then $S$** $\sim$ **if $b$ then $S$ else skip**
  - **when $b$ do $S$** $\sim$ **[await $b$; $S$]**
Syntax – Declaration

A declaration has the form

\[
\{ \langle mode \rangle \} \ variable_1, \ variable_2, \ldots, \ variable_k: \ \langle type \rangle \ \{ \text{where } \varphi \} 
\]

where the optional \( \langle mode \rangle \) is one of the following:

- **in** – Specifies variables that are input to the program/process. Cannot be modified inside the unit.
- **local** – Specifies variables that are local to the program/process but are not recognized out of it.
- **out** – Variables that are an output of the program/process. Cannot be modified outside the unit.
- **in-out** – Variables which can be modified both inside and outside the unit.

The \( \langle type \rangle \) can be a basic type which are **integer**, **natural**, **bool** (boolean) or \([L..U]\) (an integer in the range \(L..U\)).

It can also be an **array** type of the form **array** \([L..U]\) of \( \langle type \rangle \).

The optional *where* clause specifies constraints on the initial values of variables.
Syntax – Processes and Programs

A process has the form

\[
\{\langle \text{process\_name} \rangle \; :: \; \} \{\langle \text{declarations} \rangle ; \} \langle \text{statement} \rangle ; \langle \text{label} \rangle : \]

where \( \langle \text{declarations} \rangle \) are 0 or more declarations, separated by “;”. Thus, every process terminates in a label which denotes the location of control after the process has terminated. We refer to the statement as the body of the process.

A program has the form

\[
\{\langle \text{declarations} \rangle ; \} \; P_1 \| \cdots \| \; P_k,
\]

where each \( P_i, \; i = 1, \ldots, k \) is a process.
Labels

It is assumed that every statement is labeled. For a statement $S$, we define $\text{pre}(S)$ to be the preceding label which is closest to $S$ in the program.

We also define $\text{post}(S)$ inductively as follows:

- If $S; \ell :$ is the body of a process, then $\text{post}(S) = \ell$.
- If $S = [S_1; \cdots S_k]$, then $\text{post}(S_i) = \text{pre}(S_{i+1})$ for $i = 1, \ldots, k-1$ and $\text{post}(S_k) = \text{post}(S)$.
- If $S = \text{if } b \text{ then } S_1 \text{ else } S_2$ then $\text{post}(S_1) = \text{post}(S_2) = \text{post}(S)$.
- If $S = \text{while } b \text{ do } S_1$ then $\text{post}(S_1) = \text{pre}(S)$.

For a label $\ell_i$ within process $P_j$, we write $\text{at}_j\ell_i$ as an abbreviation for $\pi_j = \ell_i$.
For Example

Consider the following process:

\[
P_1 :: \begin{cases}
\ell_0 : & x := 1 \\
\ell_1 : & \textbf{while } y > 0 \textbf{ do} \\
& \begin{cases}
\ell_2 : & x := x + 2 \\
\ell_3 : & y := y - 1 \\
\end{cases} \\
\ell_4 : & x := x - 1 \\
\ell_5 : &
\end{cases}
\]

Then we have:

<table>
<thead>
<tr>
<th>\ell_0 : \cdots; \ell_1 : \cdots; \ell_4</th>
<th>post(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\ell_0 : \cdots</td>
<td>\ell_5</td>
</tr>
<tr>
<td>\ell_1 : \cdots</td>
<td>\ell_1</td>
</tr>
<tr>
<td>\ell_4 : \cdots</td>
<td>\ell_4</td>
</tr>
<tr>
<td>\ell_2 : \cdots; \ell_3 : \cdots</td>
<td>\ell_1</td>
</tr>
<tr>
<td>\ell_2 : \cdots</td>
<td>\ell_3</td>
</tr>
<tr>
<td>\ell_3 : \cdots</td>
<td>\ell_1</td>
</tr>
</tbody>
</table>
SPL: Semantics

Let $P :: declaration; P_1 || \cdots || P_k$ be a program. We proceed to construct the FDS $D_P$ corresponding to program $P$.

State Variables As the state variables, we take all the variables declared in the program and add to them a set of control variables $\pi_1, \ldots, \pi_k$.

For each $i = 1, \ldots, k$, the domain of $\pi_i$ is the set of labels appearing in process $S_i$.

For example, for program ANY-Y, the state variables are:

$$V: \begin{pmatrix} x, y & : & \text{natural} \\ \pi_1 & : & \{\ell_0, \ell_1, \ell_2\} \\ \pi_2 & : & \{m_0, m_1\} \end{pmatrix}$$

Observable Variables At this point, we take $O = V$.

Initial Condition As the initial condition, we take the conjunction of all the where clauses plus the conjunction $\pi_1 = pre(S_1) \land \cdots \land \pi_k = pre(S_k)$.

For example, the initial condition for program ANY-Y is given by

$$\Theta: \pi_1 = \ell_0 \land \pi_2 = m_0 \land x = y = 0.$$
The Transition Relation

For a subset of variables $U \subseteq V$, we denote $\text{pres}(U) = \bigwedge_{x \in U} (x' = x)$.

The transition relation $\rho$ is formed as a disjunction which standardly contains the disjunct $\rho_{\text{idle}} : \text{pres}(V)$. In addition, each statement $S$ in the program, excluding concatenation statements, contributes a disjunct $\rho_S$ according to the following recipe:

- The statement $S = \text{skip}$ in process $P_i$ contributes the disjunct

$$\pi_i = \text{pre}(S) \land \pi'_i = \text{post}(S) \land \text{pres}(V - \{\pi_i\})$$

- The statement $S = [y := e]$ in process $P_i$ contributes the disjunct

$$\pi_i = \text{pre}(S) \land \pi'_i = \text{post}(S) \land y' = e \land \text{pres}(V - \{\pi_i, y\})$$

For example, statement $\ell_1$ in program \textsc{Any-Y} contributes the disjunct

$$\pi_1 = \ell_1 \land \pi'_1 = \ell_0 \land y' = y + 1 \land \text{pres}(\{\pi_2, x\})$$
Transition Relation – Continued

- The statement $S = \textbf{await} \ b$ in process $P_i$ contributes the disjunct
  \[ \pi_i = \text{pre}(S) \land b \land \pi'_i = \text{post}(S) \land \text{pres}(V - \{\pi_i\}) \]

- The statement $S = \textbf{if} \ b \ \textbf{then} \ S_1 \ \textbf{else} \ S_2$ in process $P_i$ contributes the disjunct
  \[ \pi_i = \text{pre}(S) \land \left\{ b \land \pi'_i = \text{pre}(S_1) \lor \neg b \land \pi'_i = \text{pre}(S_2) \right\} \land \text{pres}(V - \{\pi_i\}) \]

- The statement $S = \textbf{while} \ b \ \textbf{do} \ S_1$ in process $P_i$ contributes the disjunct
  \[ \pi_i = \text{pre}(S) \land \left\{ b \land \pi'_i = \text{pre}(S_1) \lor \neg b \land \pi'_i = \text{post}(S) \right\} \land \text{pres}(V - \{\pi_i\}) \]

For example, statement $\ell_0$ of program \textsc{Any-Y} contributes the disjunct
\[ \pi_1 = \ell_0 \land \left\{ x = 0 \land \pi'_1 = \ell_1 \lor x \neq 0 \land \pi'_1 = \ell_2 \right\} \land \text{pres}(\{\pi_2, x, y\}) \]
**Justice Requirements**

Each occurrence within process $P_i$ of a statement $S$ which is a **skip**, an **assignment**, a **conditional** or a **while** statement, contributes to the justice set the requirement

$$J_S : \pi_i \neq \text{pre}(S)$$

An occurrence within $P_i$ of a statement $S = \text{await } b$, contributes the justice requirement:

$$J_S : \neg(\pi_i = \text{pre}(S) \land b).$$

For example, the justice set for program **ANY-Y** is

$$\mathcal{J} : \{\pi_1 \neq \ell_0, \pi_1 \neq \ell_1, \pi_2 \neq m_0\}$$

The implication of the **justice** requirements are:

*No statement is continuously enabled without being executed.*

or, equivalently,

*If $S$ is continuously enabled it must eventually be executed.*
Justice is not Enough. You also Need Compassion

The following program MUX-SEM, implements mutual exclusion by semaphores.

\[ y : \text{natural initially } y = 1 \]

\[
\begin{align*}
\ell_0 &: \text{ loop forever do} \\
\ell_1 &: \text{Non-critical} \\
\ell_2 &: \text{request } y \\
\ell_3 &: \text{Critical} \\
\ell_4 &: \text{release } y
\end{align*}
\] \text{ || } \begin{align*}
m_0 &: \text{ loop forever do} \\
m_1 &: \text{Non-critical} \\
m_2 &: \text{request } y \\
m_3 &: \text{Critical} \\
m_4 &: \text{release } y
\end{align*}

\[ P_1 \quad P_2 \]

The semaphore instructions request \( y \) and release \( y \) respectively stand for

\[ \langle \text{await } y > 0 ; \ y := y - 1 \rangle \text{ and } y := y + 1. \]

The compasson set of this program consists of

\[ C : \{ (at_{\ell_2} \land y > 0, \ at_{\ell_3}) , \ (at_{m_2} \land y > 0, \ at_{m_3}) \}. \]
**Program** MUX-SEM

should satisfy the following two requirements:

- **Mutual Exclusion** – No computation of the program can include a state in which process $P_1$ is at $\ell_3$ while $P_2$ is at $m_3$.
- **Accessibility** – Whenever process $P_1$ is at $\ell_2$, it shall eventually reach its critical section at $\ell_3$. Similar requirement for $P_2$.

Consider the state sequence:

$$\sigma: \langle l_0, m_0, 1 \rangle \rightarrow \ldots \rightarrow \langle l_2, m_3, 0 \rangle \xrightarrow{m_3} \langle l_2, m_4, 0 \rangle \xrightarrow{m_4} \langle l_2, m_1, 1 \rangle \xrightarrow{m_1} \langle l_2, m_2, 1 \rangle \xrightarrow{m_2}$$

which violates accessibility for process $P_1$. We should not allow this state sequence as a computation.

If the only fairness requirement associated with statement $\ell_2 : \text{request } y$ were that of justice, the above state sequence would be a computation. This is because statement $\ell_2$ is not continuously enabled. In fact, it is disabled on all states of the form $\langle l_2, m_3, 0 \rangle$. 

Compassion Saves the Day

Instead, we associate with statement \( \ell_2 \) : request \( y \) the compassion requirement

\[ (at_\ell_2 \land y > 0, \ at_\ell_3) \]

implying

Statement \( \ell_2 \) cannot be **infinitely often enabled** without being **executed**

Due to this **compassion** requirement for \( \ell_2 \), the violating state sequence is not a computation, and accessibility is guaranteed.

**Conclusion:** Justice alone is not sufficient !!!
Compassion Requirements

Each occurrence within $P_i$ of a statement $S = \text{request } y$, contributes the compassion requirement:

$$C_S : (\pi_i = \text{pre}(S) \land y > 0, \; \pi_i = \text{post}(S)).$$