SPL (Simple Programming Language)

Syntax

Basic Statements

- **skip**
- **assignment**
  \[(u_1, \ldots, u_k) := (e_1, \ldots, e_k)\]
  variables := expressions
- **await** \(c\)
  (where \(c\) is a boolean expression)
  special case: \(\text{halt} \equiv \text{await } F\)
- Communication by message-passing
  \(\alpha \leftarrow e\) (send)
  \(\alpha \Rightarrow u\) (receive)
  (where \(\alpha\) is a channel)
- Semaphore operations
  \(\text{request } r\)
  \((r > 0 \rightarrow r := r - 1)\)
  \(\text{release } r\)
  \((r := r + 1)\)
  (where \(r\) is an integer variable)

SPL (CON’T)

Schematic Statements

In Mutual-Exclusion programs:

- **noncritical**
  may not terminate
- **critical**
  terminates

In Producer-Consumer programs:

- **produce** \(x\)
  terminates – assign nonzero value to \(x\)
- **consume** \(y\)
  terminates

No program variables are modified by schematic statements. One exception:
“\(x\)” in **produce** \(x\)
SPL (CON’T)

Compound Statements

- Conditional
  if \( c \) then \( S_1 \) else \( S_2 \)
  if \( c \) then \( S \)

- Concatenation
  \( S_1; \cdots; S_k \)

Example:
when \( c \) do \( S \equiv \text{await } c; S \)

- Selection
  \( S_1 \) or \( \cdots \) or \( S_k \)

- while
  while \( c \) do \( S \)

Example:
loop forever do \( S \equiv \text{while } T \text{ do } S \)

SPL (CON’T)

Compound Statements (Con’t)

- Cooperation Statement
  \( \ell : [\ell_1: S_1; \widehat{\ell}_1: ] \parallel \cdots \parallel [\ell_k: S_k; \widehat{\ell}_k: ]; \widehat{\ell}: \) process

\( S_1, \ldots, S_k \) are parallel to one another
interleaved execution.

entry step: from \( \ell \) to \( \ell_1, \ell_2, \ldots, \ell_k \),
exit step: from \( \widehat{\ell}_1, \widehat{\ell}_2, \ldots, \widehat{\ell}_k \) to \( \widehat{\ell} \).

- Block

\[ [ \text{local declaration}; S ]; \]

local variable, \ldots, variable : type where \( \varphi_i \)

\[ \begin{align*}
y_1 &= e_1, & y_n &= e_n
\end{align*} \]

SPL (CON’T)

Basic types – boolean, integer, character, \ldots

Structured types – array, list, set, \ldots

Static variable initialization
(variables get initialized at the start of the execution)

Programs

\( P :: [\text{declaration}; P_1 :: [\ell_1: S_1; \widehat{\ell}_1: ] \parallel \cdots \parallel P_k :: [\ell_k: S_k; \widehat{\ell}_k: ] ] \)

\( P_1, \ldots, P_k \) are top-level processes
Variables in \( P \) called program variables

Declaration

mode variable, \ldots, variable: type where \( \varphi_i \)

\[ \begin{align*}
\text{in} & (\text{not modified}) & \text{constraints on} \\
\text{local} & \quad \quad \quad \text{initial values} \\
\text{out} & \quad \quad \quad \text{data-precondition of the program}
\end{align*} \]
Channel Declaration

- synchronous channels
  (no buffering capacity)
  \[\text{mode } \alpha_1, \alpha_2, \ldots, \alpha_n: \text{channel of type}\]

- asynchronous channels
  (unbounded buffering capacity)
  \[\text{mode } \alpha_1, \alpha_2, \ldots, \alpha_n: \text{channel } [1..] \text{ of type}\]

  where \(\varphi_i\)
  
  - \(\varphi_i\) is optional

  - \(\varphi_i = \Lambda\) (empty list) by default

Note: For \(\ell : [\ell_1 : S_1|| \ldots ||\ell_k : S_k]\)
\(\ell \not\sim_L \ell_1 \not\sim_L \ell_2 \not\sim_L \ldots\)
because of the entry step

Example: In Figure 0.1
\[
\ell_0 \sim_L \ell_1 \\
\ell_2 \sim_L \ell_3 \sim_L \ell_5
\]
Locations
\[ \ell \]

Identify site of control

- \[ \ell \] is the location corresponding to label \( \ell \).
- Multiple labels identifying different statements may identify the same location.
  \[ \ell = \{ \ell' \mid \ell' \sim_L \ell \} \]

Example: Fig 0.1: A fully labeled program

\[
\begin{align*}
\ell_0 &= \{ \ell_1 \} = \{ \ell_0, \ell_1 \} \\
\ell_2 &= \{ \ell_2, \ell_3, \ell_5 \} \\
\ell_4 &= \{ \ell_4 \} \\
\ell_6 &= \{ \ell_6 \} \\
\ell_7 &= \{ \ell_7 \} \\
\ell_8 &= \{ \ell_8 \}
\end{align*}
\]

Example: Fig 0.2: A partially labeled program

\[
\begin{align*}
\ell_0 &\rightarrow \ell_a^2 \\
\ell_5 &\rightarrow \ell_b^2
\end{align*}
\]

shortcut: label \( \ell_2 \) "represents" \{\( \ell_2 \), \( \ell_a^2 \), \( \ell_b^2 \)\}

Post Location
\( \ell; S; \hat{\ell} \):
\[ \text{post}(S) = [\hat{\ell}] \]

- For \[ [\ell_1; S_1; \ell_1; ] \| \cdots \| [\ell_k; S_k; \ell_k; ] \]
  \[ \text{post}(S_i) = [\hat{\ell}_i], \text{ for every } i = 1, \ldots, k \]

- For \[ S = [\ell_1; S_1; \ldots; \ell_k; S_k] \]
  \[ \text{post}(S_i) = [\hat{\ell}_{i+1}], \text{ for } i = 1, \ldots, k-1 \]
  \[ \text{post}(S_k) = \text{post}(S) \]

- For \[ S = [\ell_1; S_1 \text{ or } \ldots \text{ or } \ell_k; S_k] \]
  \[ \text{post}(S_1) = \cdots = \text{post}(S_k) = \text{post}(S) \]

- For \[ S = [\text{if } c \text{ then } S_1 \text{ else } S_2] \]
  \[ \text{post}(S_1) = \text{post}(S_2) = \text{post}(S) \]

- For \[ [\ell; \text{ while } c \text{ do } S'] \]
  \[ \text{post}(S') = [\ell] \]
Ancestor

S is an ancestor of S′ if S′ is a substatement of S.

S is a common ancestor of S1 and S2 if it is an ancestor of both S1 and S2.

S is a least common ancestor (LCA) of S1 and S2 if S is a common ancestor of S1 and S2 and any other common ancestor of S1 and S2 is an ancestor of S.

LCA is unique for given statements S1 and S2.

Example: \([S_1; [S_2∥S_3]; S_4]∥ S_5\)

<table>
<thead>
<tr>
<th>LCA of S2, S3</th>
<th>[S_2∥S_3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCA of S2, S4</td>
<td>[S_1; [S_2∥S_3]; S_4]</td>
</tr>
<tr>
<td>LCA of S2, S5</td>
<td>[S_1; [S_2∥S_3]; S_4]∥ S_5</td>
</tr>
</tbody>
</table>

Parallel Labels

- Statements S and ˜S are parallel if their LCA is a cooperation statement that is different from statements S and ˜S.

Example: S = \([S_1; [S_2∥S_3]; S_4]∥ S_5\)

<table>
<thead>
<tr>
<th>Statements</th>
<th>LCA</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_2 parallel to S_3</td>
<td>S_2∥ S_3</td>
</tr>
<tr>
<td>S_2 parallel to S_5</td>
<td>S</td>
</tr>
<tr>
<td>S_2 not parallel to S_4</td>
<td>[S_1; · · ·; S_4] not coop.</td>
</tr>
<tr>
<td>S_2 not parallel to S_2∥ S_3</td>
<td>S_2∥ S_3 same</td>
</tr>
</tbody>
</table>

- parallel labels – labels of parallel statements

Conflicting Labels

conflicting labels – not equivalent and not parallel

Example:
\[
\begin{align*}
\ell_1 &: S_1; \\
\ell_2 &: ([\ell_3: S_3; ˜\ell_3:]||[\ell_4: S_4; ˜\ell_4:]); || [\ell_5: S_5; ˜\ell_5: ]
\end{align*}
\]

\(\ell_3\) is parallel to each of \(\{\ell_4, ˜\ell_4, \ell_6, ˜\ell_6\}\) and in conflict with each of \(\{\ell_1, \ell_2, ˜\ell_3, \ell_5, ˜\ell_5\}\).

\(\ell_6\) and ˜\(\ell_6\) are in conflict with each other but are parallel to each of \(\{\ell_1, \ell_2, ˜\ell_3, \ell_4, ˜\ell_4, \ell_5, ˜\ell_5\}\)

Critical References

Writing References:
\[
\begin{align*}
x &: \cdots \alpha \Rightarrow u \text{ produce } x \text{ request } r \\
\uparrow & \uparrow \uparrow \uparrow \uparrow \uparrow \\
\text{release } r \\
\uparrow
\end{align*}
\]

Reading References: all other references

- critical reference of a variable in S if:
  - writing ref to a variable that has reading or writing refs in \(S′\) (parallel to S)
  - reading reference to a variable that has writing references in \(S′\) (parallel to S)
  - reference to a channel
Limited Critical References (LCR)

Statement obeys LCR restriction (LCR-Statement) if each test (for await, conditional, while) and entire statement (for assignment) contains at most one critical reference.

Example: Fig 0.3
\(\ell_2, m_1, m_3\) are LCR-Statements
\(\ell_1, m_2\) violate the LCR-requirement

LCR-Program: only LCR-statements

Interleaved vs. Concurrent Execution

Claim: If \(P\) is an LCR program, then the interleaving computations of \(P\) and the concurrent executions of \(P\) give the same results.

Discussion & explanation: Blue Book.

SPL Semantics

Transition Semantics:

\[
\begin{align*}
\text{SPL } P & \quad \text{computation of } P \\
\downarrow & \quad \uparrow \\
\text{FTS } \Phi & \quad \text{computation of } \Phi
\end{align*}
\]

Given an SPL-program \(P\), we can construct the corresponding FTS \(\Phi = \langle V, \Theta, T, J, C \rangle\):

- system variables \(V\)
  \(Y = \{y_1, \ldots, y_n\}\) – program variables of \(P\)
  domains: as declared in \(P\)
  \(\pi\) – control variable
  domain: sets of locations in \(P\)
  \(V = Y \cup \{\pi\}\)

SPL Semantics (Con’t)

Comments:

- For label \(\ell\), at \(\ell\): \([\ell] \in \pi\)
  at' \(\ell\): \([\ell] \in \pi'\)

Note: When going from an SPL program to an FTS we lose the sequential nature of the program. We need to model control explicitly in the FTS: \(\pi\) can be viewed as a program counter.
SPL Semantics (Con’t)

Example: Fig 0.1

\[ V = \{ \pi, a, b, y_1, y_2, g \} \]
\[ \pi \text{ - ranges over subsets of } \{ [\ell_1], [\ell_2], [\ell_4], [\ell_6], [\ell_7], [\ell_8] \} \]
\[ a, b, \ldots, g \text{ - range over integers} \]

- Initial Condition \( \Theta \)
  For \( P :: \) \[ \text{dec; } P_1 :: [\ell_1; S_1; \hat{\ell}_1: ] \| \cdots \| P_k :: [\ell_k; S_k; \hat{\ell}_k: ] ] \]
  with data-precondition \( \varphi \),
  \[ \Theta: \pi = \{ [\ell_1], \ldots, [\ell_k] \} \land \varphi \]

Example: Fig 0.1
\[ \Theta: \pi = \{ [\ell_1] \} \land \]
\[ a > 0 \land b > 0 \land y_1 = a \land y_2 = b \]
data-precondition

SPL Semantics (Con’t)

- Transitions \( \mathcal{T} \)
  \[ \mathcal{T} = \{ \tau_I \} \cup \left\{ \text{transitions associated with the statements of } P \right\} \]
  where \( \tau_I \) is the “idling transition”
  \[ \rho_{\tau_I}: V' = V \]

abbreviation
- \( \text{pres}(U): \bigwedge_{u \in U} (u' = u) \) (where \( U \subseteq V \))
  the value of \( u \in U \) are preserved
- \( \text{move}(L, \hat{L}): L \subseteq \pi \land \pi' = (\pi - L) \cup \hat{L} \)
  where \( L, \hat{L} \) are sets of locations
- \( \text{move}(\ell, \hat{\ell}): \text{move}(\{[\ell]\}, \{[\hat{\ell}]\}) \)

in \( a, b : \text{integer where } a > 0, b > 0 \)
local \( y_1, y_2: \text{integer where } y_1 = a, y_2 = b \)
out \( g : \text{integer} \)

[\ell_1: \text{while } y_1 \neq y_2 \text{ do}
\begin{align*}
\ell_2: & \quad [\ell_2': \text{await } y_1 > y_2; \ell_4: y_1 := y_1 - y_2] \\
\ell_3: & \quad [\ell_3': \text{await } y_2 > y_1; \ell_6: y_2 := y_2 - y_1] \\
\ell_7: & \quad g := y_1 \\
\ell_8: & \quad \end{align*}
]

Figure 0.2

A Partially Labeled Program GCD

SPL Semantics (Con’t)

We list the transitions (transition relations) associated with the statements of \( P \)

\[ \ell : S \quad P \ell \]

Basic Statements

\[ \ell: \text{skip}; \hat{\ell}: \rightarrow \text{move}(\ell, \hat{\ell}) \land \text{pres}(Y) \]
\[ \ell: \pi := \nu; \hat{\ell}: \rightarrow \text{move}(\ell, \hat{\ell}) \land \pi' = \nu \land \text{pres}(Y - \{\pi\}) \]
SPL Semantics (Con’t)

Basic Statements (Con’t)

ℓ: await c; ˆℓ:  → move(ℓ, ˆℓ) ∧ c ∧ pres(Y )

ℓ: request r; ˆℓ:  → move(ℓ, ˆℓ) ∧ r > 0
∧ r′ = r − 1
∧ pres(Y − {r})

ℓ: release r; ˆℓ:  → move(ℓ, ˆℓ) ∧ r′ = r + 1
∧ pres(Y − {r})

Schematic Statements ρℓ

ℓ: noncritical; ˆℓ:  → move(ℓ, ˆℓ) ∧ pres(Y)
(nontermination modeled by τℓ /∈ J)

ℓ: critical; ˆℓ:  → move(ℓ, ˆℓ) ∧ pres(Y)

Compound Statements

ℓ: [if c then ℓ1:S1 else ℓ2:S2]; ˆℓ:  → 
ρℓ: ρTℓ ∨ ρFℓ where

ρTℓ: move(ℓ, ℓ1) ∧ c ∧ pres(Y)
ρFℓ: move(ℓ, ℓ2) ∧ ¬c ∧ pres(Y)

ℓ: [while c do [ℓ: S]]; ˆℓ:  → 
ρℓ: ρTℓ ∨ ρFℓ where

ρTℓ: move(ℓ, ℓ) ∧ c ∧ pres(Y)
ρFℓ: move(ℓ, ℓ) ∧ ¬c ∧ pres(Y)

ℓ: [[ℓ1:S1; ˆℓ1]; . . . ; [ℓk:Sk; ˆℓk]]; ˆℓ:  → 
ρ£ℓ: move(ℓ, ℓ1, . . . , ℓk) ∧ pres(Y) (entry)
ρ£xℓ: move(ℓ1, . . . , ℓk, ˆℓ) ∧ pres(Y) (exit)
Grouped Statements \( \langle S \rangle \)
executed in a single atomic step

**Example:**
\[ \langle x := y + 1; z := 2x + 1 \rangle \]
\[ x' = y + 1 \land z' = 2y + 3 \]
the same as \( \langle x, z \rangle := (y + 1, 2y + 3) \)

**Example:**
\[ \langle a := 3; a := 5 \rangle \]
a' = 5
\[ a = 3 \] is never visible to the outside world, nor to other processes

---

**SPL Semantics (Con't)**

- **Justice Set \( \mathcal{J} \)**
  All transitions except \( \tau_I \) and all transitions associated with noncritical statements

- **Compassion Set \( \mathcal{C} \)**
  All transitions associated with send, receive, request statements

---

**Computations of Programs**

local \( x \): integer where \( x = 1 \)

\[ P_1 :: \begin{cases} \ell_0: & \begin{cases} \ell_0^c: \text{await } x = 1 \\ \ell_0^d: \text{skip} \end{cases} \end{cases} \]
\( \parallel P_2 :: \begin{cases} m_0: \text{while } \top \text{ do } \\
\quad \begin{cases} m_1: x := -x \end{cases} \end{cases} \]

\[ \ell_1: \]

**Fig 0.4** Process \( P_1 \) terminates in all computations.

\[ \begin{align*}
\sigma: & \langle \pi: \{\ell_0, m_0\}, x: 1 \rangle \xrightarrow{m_0} \langle \pi: \{\ell_0, m_1\}, x: 1 \rangle \xrightarrow{m_1} \\
& \langle \pi: \{\ell_0, m_0\}, x: -1 \rangle \xrightarrow{m_0} \langle \pi: \{\ell_0, m_1\}, x: -1 \rangle \xrightarrow{m_1} \\
& \langle \pi: \{\ell_0, m_0\}, x: 1 \rangle \xrightarrow{m_0} \ldots \\
\end{align*} \]

\( \sigma \) is **not** a computation. Unjust towards \( \ell_0^d \)
(enabled on all states but never taken)

---

**Computations of Programs (Con't)**

local \( x \): integer where \( x = 1 \)

\[ P_1 :: \begin{cases} \ell_0: & \begin{cases} \ell_0^c: \text{await } x = 1 \\ \ell_0^d: \text{skip} \end{cases} \end{cases} \]
\( \parallel P_2 :: \begin{cases} m_0: \text{while } \top \text{ do } \\
\quad \begin{cases} m_1: x := -x \end{cases} \end{cases} \]

\[ \ell_1: \]

**Fig 0.5** \( \text{skip } \rightarrow \text{ await } x \neq 1 \)

\[ \begin{align*}
\sigma: & \langle \pi: \{\ell_0, m_0\}, x: 1 \rangle \xrightarrow{m_0} \langle \pi: \{\ell_0, m_1\}, x: 1 \rangle \xrightarrow{m_1} \\
& \langle \pi: \{\ell_0, m_0\}, x: -1 \rangle \xrightarrow{m_0} \langle \pi: \{\ell_0, m_1\}, x: -1 \rangle \xrightarrow{m_1} \\
& \langle \pi: \{\ell_0, m_0\}, x: 1 \rangle \xrightarrow{m_0} \ldots \\
\end{align*} \]

\( \sigma \) is a computation –
since none of the just transitions are continually enabled.
Computations of Programs (Con’t)

local x: integer where x = 1

\[
P_1 ::
\begin{align*}
\ell_0: & \text{ if } x = 1 \text{ then} \\
\ell_1: & \text{ skip} \\
\ell_2: & \text{ skip} \\
\ell_3: & \text{ skip}
\end{align*}
\]

Fig 0.6 Process \(P_1\) terminates in all computations.

\(\sigma: \langle \pi: \{\ell_0, m_0\}, x: 1 \rangle \xrightarrow{m_0} \langle \pi: \{\ell_0, m_1\}, x: 1 \rangle \xrightarrow{m_1}
\)

\(\langle \pi: \{\ell_0, m_0\}, x: -1 \rangle \xrightarrow{m_0} \langle \pi: \{\ell_0, m_1\}, x: -1 \rangle \xrightarrow{m_1}
\)

\(\langle \pi: \{\ell_0, m_0\}, x: 1 \rangle \xrightarrow{m_0} \ldots
\)

\(\sigma\) is not a computation – since \(\ell_0\) is continually enabled, but not taken.

SPL Semantics (Con’t)

accessible configuration – appears as value of \(\pi\) in some accessible state

Example:

\(\{\ell_0, [m_1]\}\) does not appear in any accessible state

The Mutual-Exclusion Problem

loop forever do

\[
\begin{array}{c}
\text{noncritical} \\
\cdots \\
\text{critical} \\
\cdots
\end{array}
\]

loop forever do

\[
\begin{array}{c}
\text{noncritical} \\
\cdots \\
\text{critical} \\
\cdots
\end{array}
\]

Requirements:

- **Exclusion**
  While one of the processes is in its critical section, the other is not

- **Accessibility**
  Whenever a process is at the noncritical section exit, it must eventually reach its critical section

Example: mutual exclusion by semaphores

Fig. 0.7
local $y$: integer where $y = 1$

$$
\begin{align*}
\ell_0: \text{loop forever do} \\
&\begin{cases}
\ell_1: \text{noncritical} \\
\ell_2: \text{request } y \\
\ell_3: \text{critical} \\
\ell_4: \text{release } y
\end{cases} \\
\end{align*}
$$

$$
\begin{align*}
m_0: \text{loop forever do} \\
&\begin{cases}
m_1: \text{noncritical} \\
m_2: \text{request } y \\
m_3: \text{critical} \\
m_4: \text{release } y
\end{cases} \\
\end{align*}
$$

Fig. 0.7 Program MUX-SEM

Message-Passing Programs

Example: Producer-Consumer

assumption:

channel send $\leq N$ values

local $\text{send}, \text{ack}$: channel $[1..N]$ of integer

where $\text{send} = \Lambda$, $\text{ack} = [1, \ldots, 1]$

$$
\begin{align*}
\text{Prod} :: \\
&\begin{cases}
\ell_1: \text{produce } x \\
\ell_5: \text{ack } \to t \\
\ell_5: \text{send } \to z
\end{cases} \\
\end{align*}
$$

$$
\begin{align*}
\text{Cons} :: \\
&\begin{cases}
m_1: \text{send } \to y \\
m_5: \text{ack } \leftarrow 1 \\
m_5: \text{consume } y
\end{cases} \\
\end{align*}
$$

Fig. 0.9 Program PROD-CONS