Shared-Variable Concurrency

Xinyu Feng

University of Science and Technology of China

12/17/2013
Parallel Composition (or Concurrency Composition)

Syntax:

\[(\text{comm}) \ c ::= \ldots \mid c_0 \parallel c_1 \mid \ldots\]

Note we allow nested parallel composition, e.g.,
\[(c_0; (c_1 \parallel c_2)) \parallel c_3.\]

Operational Semantics:

\[
\begin{align*}
(c_0, \sigma) & \rightarrow (c'_0, \sigma') \\
(c_0 \parallel c_1, \sigma) & \rightarrow (c'_0 \parallel c_1, \sigma') \\
(Skip \parallel Skip, \sigma) & \rightarrow (Skip, \sigma) \\
(c_1, \sigma) & \rightarrow (c'_1, \sigma') \\
(c_0 \parallel c_1, \sigma) & \rightarrow (c_0 \parallel c'_1, \sigma') \\
(c_i, \sigma) & \rightarrow (\text{abort}, \sigma'), \quad i \in \{0, 1\} \\
(c_0 \parallel c_1, \sigma) & \rightarrow (\text{abort}, \sigma')
\end{align*}
\]

We have to use small-step semantics (instead of big-step semantics) to model concurrency.
Interference

Example:

\[
\begin{align*}
y &:= x + 1; & y &:= x + 1; \\
x &:= y + 1 & x &:= x + 1
\end{align*}
\]

Suppose initially \( \sigma_x = \sigma_y = 0 \). What are the possible results?

(1) \( y = 1, x = 2; \) (2) \( y = 1, x = 3; \) (3) \( y = 3, x = 3; \) (4) \( y = 2, x = 3 \)

Two commands \( c_0 \) and \( c_1 \) are said to interfere if:

\[
(fv(c_0) \cap fa(c_1)) \cup (fv(c_1) \cap fa(c_0)) \neq \emptyset
\]

If \( c_0 \) and \( c_1 \) interfere, we say there are race conditions (or races) in \( c_0 \parallel c_1 \).

When \( c_0 \) and \( c_1 \) do not interfere, nor terminate by failure, the concurrent composition \( c_0 \parallel c_1 \) is determinate.
A benign race:

\[ k := -1; \]

\[
\begin{align*}
&\text{(newvar } i := 0 \text{ in while } i \leq n \land k = -1 \text{ do} } \\
&\quad \text{if } f(i) \geq 0 \text{ then } k := i \text{ else } i := i + 2 \\
\end{align*}
\]

\[
\| \text{newvar } i := 1 \text{ in while } i \leq n \land k = -1 \text{ do} \\
\quad \text{if } f(i) \geq 0 \text{ then } k := i \text{ else } i := i + 2 \\
\]

A problematic version:

\[ k := -1; \]

\[
\begin{align*}
&\text{(newvar } i := 0 \text{ in while } i \leq n \land k = -1 \text{ do} } \\
&\quad \text{if } f(i) \geq 0 \text{ then print}(i) ; \text{print}(f(i)) \text{ else } i := i + 2 \\
\end{align*}
\]

\[
\| \text{newvar } i := 1 \text{ in while } i \leq n \land k = -1 \text{ do} \\
\quad \text{if } f(i) \geq 0 \text{ then print}(i) ; \text{print}(f(i)) \text{ else } i := i + 2 \\
\]
Conditional Critical Regions

We could use a critical region to achieve mutual exclusive access of shared variables.

Syntax:

$$( \text{comm} ) \ c \ ::= \ \text{await } b \ \text{then } \hat{c}$$

where $\hat{c}$ is a sequential command (a command with no await and parallel composition).

Semantics:

$$[[ b ]]_{\text{boolexp } \sigma} = \text{true} \quad (\hat{c}, \sigma) \longrightarrow^* (\text{Skip}, \sigma')$$

$$(\text{await } b \ \text{then } \hat{c}, \sigma) \longrightarrow (\text{Skip}, \sigma')$$

$$[[ b ]]_{\text{boolexp } \sigma} = \text{false}$$

$$(\text{await } b \ \text{then } \hat{c}, \sigma) \longrightarrow (\text{Skip} ; \text{await } b \ \text{then } \hat{c}, \sigma)$$

The second rule gives us a “busy-waiting” semantics. If we eliminate that rule, the thread will be blocked when the condition does not hold.
Achieving Mutual Exclusion

\[ k := -1; \]

\[
\begin{align*}
&\text{newvar } i := 0 \text{ in while } i \leq n \land k = -1 \text{ do} \\
&\quad (\text{if } f(i) \geq 0 \text{ then (await busy = 0 then busy := 1); print}(i) ; \text{print}(f(i)) ; \text{busy := 0} \\
&\quad \text{else } i := i + 2) \\
\end{align*}
\]

\[
\begin{align*}
&\text{newvar } i := 1 \text{ in while } i \leq n \land k = -1 \text{ do} \\
&\quad (\text{if } f(i) \geq 0 \text{ then (await busy = 0 then busy := 1); print}(i) ; \text{print}(f(i)) ; \text{busy := 0} \\
&\quad \text{else } i := i + 2))
\end{align*}
\]
A syntactic sugar:

\[
\text{atomic}\{c\} \overset{\text{def}}{=} \text{await true then } c
\]

We may also use the short-hand notation \(\langle c \rangle\).

Semantics:

\[
\begin{align*}
(c, \sigma) \longrightarrow^* (\text{Skip}, \sigma') \\
(\text{atomic}\{c\}, \sigma) \longrightarrow (\text{Skip}, \sigma')
\end{align*}
\]

It gives the programmer control over the size of atomic actions.

Reynolds uses \texttt{crit}\ c instead of \texttt{atomic}\{c\}. 

---

**Xinyu Feng**
**Shared-Variable Concurrency**
await busy0 = 0
  then busy0 := 1;
 await busy1 = 0
  then busy1 := 1;

...
Fairness

\[ k := -1; \]
\[
\begin{aligned}
    & (\text{newvar } i := 0 \text{ in while } k = -1 \text{ do} \\
    & \quad \text{if } f(i) \geq 0 \text{ then } k := i \text{ else } i := i + 2 \\
    & ) \]
\[
\begin{aligned}
    & \parallel \text{newvar } i := 1 \text{ in while } k = -1 \text{ do} \\
    & \quad \text{if } f(i) \geq 0 \text{ then } k := i \text{ else } i := i + 2
\end{aligned}
\]

Suppose \( f(i) < 0 \) for all even number \( i \). Then there's an infinite execution in the form of:

\[ \ldots \rightarrow (c_1 \parallel c', \sigma_1) \rightarrow (c_2 \parallel c', \sigma_2) \rightarrow \ldots \rightarrow (c_n \parallel c', \sigma_n) \rightarrow \ldots \]

An execution of concurrent processes is \textit{unfair} if it does not terminate but, after some finite number of steps, there is an unterminated process that never makes a transition.
A fair execution of the following program would always terminate:

\[
\text{newvar } y := 0 \text{ in } (x := 0;((\text{while } y = 0 \text{ do } x := x + 1) \parallel y := 1))
\]

Stronger fairness is needed to rule out infinite execution of the following program:

\[
\text{newvar } y := 0 \text{ in }
\begin{align*}
& (x := 0; \\
& (\text{while } y = 0 \text{ do } x := 1 - x) \parallel (\text{await } x = 1 \text{ then } y := 1))
\end{align*}
\]
Trace Semantics

Can we give a denotational semantics to concurrent programs? The domain-based approach is complex. Here we use transition traces to model the execution of programs.

Execution of \((c_0, \sigma_0)\) in a concurrent setting:

\[
(c_0, \sigma_0) \rightarrow (c_1, \sigma'_0), (c_1, \sigma_1) \rightarrow (c_2, \sigma'_1), \ldots, (c_{n-1}, \sigma_{n-1}) \rightarrow (\text{Skip}, \sigma'_{n-1})
\]

The gap between \(\sigma'_i\) and \(\sigma_{i+1}\) reflects the intervention of the environment (other threads).

It could be infinite if \((c_0, \sigma_0)\) does not terminate:

\[
(c_0, \sigma_0) \rightarrow (c_1, \sigma_1), (c_1, \sigma'_1) \rightarrow (c_2, \sigma_2), \ldots
\]

We omit the commands to get a transition trace:

\[
(\sigma_0, \sigma'_0), (\sigma_1, \sigma'_1), \ldots, (\sigma_{n-1}, \sigma'_{n-1})
\]

or

\[
(\sigma_0, \sigma'_0), (\sigma_1, \sigma'_1), \ldots
\]
A trace \((\sigma_0, \sigma'_0), (\sigma_1, \sigma'_1), \ldots, (\sigma_{n-1}, \sigma'_{n-1})\) (or \((\sigma_0, \sigma'_0), (\sigma_1, \sigma'_1), \ldots\)) is said to be \textit{Interference-Free} iff \(\forall i. \sigma'_i = \sigma_{i+1}\).
We use $\tau$ to represent individual transition traces, and $\mathcal{T}$ for a set of traces.

- $\epsilon$ empty trace
- $\tau_1 \mathbin{++} \tau_2 \overset{\text{def}}{=} \text{concatenation of } \tau_1 \text{ and } \tau_2$
  - $\tau_1$ if $\tau_1$ is infinite.

- $\mathcal{T}_1 \mathbin{;} \mathcal{T}_2 \overset{\text{def}}{=} \{\tau_1 \mathbin{++} \tau_2 \mid \tau_1 \in \mathcal{T}_1 \text{ and } \tau_2 \in \mathcal{T}_2\}$

- $\mathcal{T}^0 \overset{\text{def}}{=} \{\epsilon\}$

- $\mathcal{T}^{n+1} \overset{\text{def}}{=} \mathcal{T} \mathbin{;} \mathcal{T}^n$

- $\mathcal{T}^* \overset{\text{def}}{=} \bigcup_{n=0}^{\infty} \mathcal{T}^n$

- $\mathcal{T}^\omega \overset{\text{def}}{=} \{\tau_0 \mathbin{++} \tau_1 \mathbin{++} \ldots \mid \tau_i \in \mathcal{T}\}$

Note the difference between $\mathcal{T}^*$ and $\mathcal{T}^\omega$. 
\( \mathcal{T}[x := e] = \{(\sigma, \sigma') \mid \sigma' = \sigma[x \leadsto \llbracket e \rrbracket_{\text{intexp}} \sigma] \} \)

\( \mathcal{T}[\text{Skip}] = \{(\sigma, \sigma) \mid \sigma \in \Sigma\} \)

\( \mathcal{T}[c_0 ; c_1] = \mathcal{T}[c_0] ; \mathcal{T}[c_1] \)

\( \mathcal{T}[\text{if } b \text{ then } c_1 \text{ else } c_2] = (\mathcal{B}[b] ; \mathcal{T}[c_1]) \cup (\mathcal{B}[\neg b] ; \mathcal{T}[c_2]) \)

where \( \mathcal{B}[b] = \{(\sigma, \sigma) \mid \llbracket b \rrbracket_{\text{boolexp}} \sigma = \text{true}\} \)

\( \mathcal{T}[\text{while } b \text{ do } c] = ((\mathcal{B}[b] ; \mathcal{T}[c])^* ; \mathcal{B}[\neg b]) \cup (\mathcal{B}[b] ; \mathcal{T}[c])^\omega \)
How to give semantics to `newvar x := e in c`?

**Definition:** `local-global(x, e, \tau, \hat{\tau})` iff the following are true (suppose `\tau = (\sigma_0, \sigma'_0), (\sigma_1, \sigma'_1), \ldots` and `\hat{\tau} = (\hat{\sigma}_0, \hat{\sigma}'_0), (\hat{\sigma}_1, \hat{\sigma}'_1), \ldots`):

- They have the same length;
- For all `x' \neq x`, `\sigma_i x = \hat{\sigma}_i x` and `\sigma'_i x = \hat{\sigma}'_i x`;
- For all `i`, `\sigma_{i+1} x = \sigma'_i x`;
- For all `i`, `\hat{\sigma}_i x = \hat{\sigma}'_i x`;
- `\sigma_0 x = [e]_{\text{intexp}} \hat{\sigma}_0`.

\[
\mathcal{T}[\text{newvar x := e in c}] = \{\hat{\tau} | \tau \in \mathcal{T}[c] \text{ and } local-global(x, e, \tau, \hat{\tau})\}
\]
We view a trace $\tau$ as a function mapping indices to the corresponding transitions.

**Definition:** $\text{fair-merge}(\tau_1, \tau_2, \tau)$ iff there exist functions $f \in \text{dom}(\tau_1) \rightarrow \text{dom}(\tau)$ and $g \in \text{dom}(\tau_2) \rightarrow \text{dom}(\tau)$ such that the following are true:

- $f$ and $g$ are monotone injections:
  $$i < j \implies (f \ i < f \ j) \land (g \ i < g \ j)$$

- $\text{ran}(f) \cap \text{ran}(g) = \emptyset$ and $\text{ran}(f) \cup \text{ran}(g) = \text{dom}(\tau)$;

- $\forall i. \tau_1(i) = \tau(f \ i) \land \tau_2(i) = \tau(g \ i)$

Then $\mathcal{T}_{\text{fair}}[c_1 \parallel c_2] =$
$$\{ \tau \mid \exists \tau_1 \in \mathcal{T}_{\text{fair}}[c_1], \tau_2 \in \mathcal{T}_{\text{fair}}[c_2]. \text{fair-merge}(\tau_1, \tau_2, \tau) \}$$
Definition: unfair-merge(\(\tau_1, \tau_2, \tau\)) if one of the following are true:

- fair-merge(\(\tau_1, \tau_2, \tau\))
- \(\tau_1\) is infinite and there exist \(\tau'_2\) and \(\tau''_2\) such that \(\tau_2 = \tau'_2+\tau''_2\) and fair-merge(\(\tau_1, \tau'_2, \tau\))
- \(\tau_2\) is infinite, and there exist \(\tau'_1\) and \(\tau''_1\) such that \(\tau_1 = \tau'_1+\tau''_1\) and fair-merge(\(\tau'_1, \tau_2, \tau\))

\[
\mathcal{T}_{\text{unfair}}[c_1 \parallel c_2] = \{\tau \mid \exists \tau_1 \in \mathcal{T}_{\text{unfair}}[c_1], \tau_2 \in \mathcal{T}_{\text{unfair}}[c_2]. \text{unfair-merge}(\tau_1, \tau_2, \tau)\}
\]
$$\mathcal{T}[\text{await } b \text{ then } c] =$$

$$\left( B[\neg b] ; \mathcal{T}[\text{Skip}] \right)^* ;$$

$$\{ (\sigma, \sigma') \mid [b]_{\text{boolexp}} \sigma = \text{true}$$

and there exist $\sigma'_0, \sigma_1, \sigma'_1, \ldots, \sigma_n$ such that

$$(\sigma, \sigma'_0), (\sigma_1, \sigma'_1), \ldots, (\sigma_n, \sigma') \in \mathcal{T}[c]$$

and it is Interference-Free.$$ \}$$

$$\cup \left( B[\neg b] ; \mathcal{T}[\text{Skip}] \right)^\omega$$
The semantics is equivalent to the following:

\[ \mathcal{T}[\llbracket c \rrbracket] \triangleq \]

\[ \{ (\sigma_0, \sigma'_0), \ldots, (\sigma_n, \sigma'_n) \mid \]
\[ \text{there exist } c_0, \ldots, c_n \text{ such that } c_0 = c, \]
\[ \forall i \in [0, n - 1]. \ (c_i, \sigma_i) \rightarrow (c_{i+1}, \sigma'_i), \]
\[ \text{and } (c_n, \sigma_n) \rightarrow (\text{Skip}, \sigma'_n) \} \]
\[ \cup \{ (\sigma_0, \sigma'_0), (\sigma_1, \sigma'_1), \ldots \mid \]
\[ \text{there exist } c_0, c_1, \ldots \text{ such that } c_0 = c, \]
\[ \text{and for all } i, (c_i, \sigma_i) \rightarrow (c_{i+1}, \sigma'_i) \} \]
The trace semantics we just defined is not abstract enough. It distinguishes the following programs (which should be viewed equivalent):

\[
x := x + 1 \\
x := x + 1 ; \text{Skip} \\
\text{Skip} ; x := x + 1
\]

Also consider the following two programs:

\[
x := x + 1 ; x := x + 1 \\
(x := x + 1 ; x := x + 1) \text{ choice } x := x + 2
\]
Stuttering and Mumbling

\[
\frac{\tau \preceq \tau}{(\sigma, \sigma'), (\sigma', \sigma''), \tau \prec (\sigma, \sigma''), \tau}
\]

\[
\frac{\tau \prec \tau'}{\tau \prec \tau''}
\]

\[
\frac{\tau \prec \tau' \quad \tau' \prec \tau''}{\tau \prec \tau''}
\]

\[
\frac{\tau \prec \tau'}{(\sigma, \sigma'), \tau \prec (\sigma, \sigma'), \tau'}
\]

\[
\mathcal{T}^\dagger \overset{\text{def}}{=} \{ \tau \mid \tau \in \mathcal{T} \text{ or } \exists \tau' \in \mathcal{T}. \tau' \prec \tau \}
\]

\[
\mathcal{T}^\ast[\mathbb{C}] \overset{\text{def}}{=} (\mathcal{T}[\mathbb{C}])^\dagger
\]
The new semantics $T^*[c]$ is equivalent to the following:

$$T[c] \overset{\text{def}}{=} \{(\sigma_0, \sigma'_0), \ldots, (\sigma_n, \sigma'_n) \mid$$
$$\text{there exist } c_0, \ldots, c_n \text{ such that } c_0 = c,$$
$$\forall i \in [0, n-1]. (c_i, \sigma_i) \rightarrow^* (c_{i+1}, \sigma'_i),$$
$$\text{and } (c_n, \sigma_n) \rightarrow^* (\text{Skip}, \sigma'_n)\}$$
$$\cup\{(\sigma_0, \sigma'_0), (\sigma_1, \sigma'_1), \ldots \mid$$
$$\text{there exist } c_0, c_1, \ldots \text{ such that } c_0 = c,$$
$$\forall i. (c_i, \sigma_i) \rightarrow^* (c_{i+1}, \sigma'_i),$$
$$\text{and for infinitely many } i \geq 0, (c_i, \sigma_i) \rightarrow^+ (c_{i+1}, \sigma'_i)\}$$