Concurrent Execution

Abhik Roychoudhury
CS 3211
National University of Singapore

Modified from Kramer andMagee’s lecture notes.
Reading material: Chapter 3 of Textbook.

1. Concurrent processes
We structure complex systems as sets of simpler activities, each represented as a sequential process. Processes can overlap or be concurrent, so as to reflect the concurrency inherent in the physical world, or to offload time-consuming tasks, or to manage communications or other devices.

Describing concurrent software can be complex and error prone. A rigorous engineering approach is essential.

2. Definitions
- **Concurrency**: Logically simultaneous processing. Does not imply multiple processing elements (PEs). Requires interleaved execution on a single PE.
- **Parallelism**: Physically simultaneous processing. Involves multiple PEs and/or independent device operations.

Both concurrency and parallelism require controlled access to shared resources. We use the terms parallel and concurrent interchangeably and generally do not distinguish between real and pseudo-concurrent execution.

3.1 Modeling Concurrency
- How should we model process execution speed?
  - arbitrary speed (we abstract away time)
- How do we model concurrency?
  - arbitrary relative order of actions from different processes (interleaving but preservation of each process order)
- What is the result?
  - provides a general model independent of scheduling (asynchronous model of execution)

parallel composition - action interleaving

If P and Q are processes then (P||Q) represents the concurrent execution of P and Q. The operator || is the parallel composition operator.

ITCH = (scratch→STOP) long
CONVERSE = (think→talk→STOP) long

||CONVERSE_ITCH = (ITCH || CONVERSE) long

think→talk→scratch think→scratch→think
Possible traces as a result of action interleaving.

parallel composition - action interleaving

ITCH = (scratch→STOP) long
CONVERSE = (think→talk→STOP) long

||CONVERSE_ITCH = (ITCH || CONVERSE) long

think→talk→scratch think→scratch→think
Possible traces as a result of action interleaving.
parallel composition - algebraic laws

| Commutative: | (P||Q) = (Q||P) |
| Associative: | (P||(Q||R)) = ((P||Q)||R) = (P||Q||R). |

Clock radio example:

CLOCK = (tick->CLOCK).
RADIO = (on->off->RADIO).
||CLOCK_RADIO = (CLOCK || RADIO).

modeling interaction - shared actions

If processes in a composition have actions in common, these actions are said to be shared. Shared actions are the way that process interaction is modelled. While unshared actions may be arbitrarily interleaved, a shared action must be executed at the same time by all processes that participate in the shared action.

MAKER = (make->ready->MAKER).
USER = (ready->use->USER).
||MAKER_USER = (MAKER || USER).

modeling interaction - handshake

A handshake is an action acknowledged by another:

MAKERv2 = (make->ready->used->MAKERv2).
USERv2 = (ready->use->used ->USERv2).
||MAKER_USERv2 = (MAKERv2 || USERv2).

modeling interaction - multiple processes

Multi-party synchronization:

MAKE_A = (makeA->ready->used->MAKE_A).
MAKE_B = (makeB->ready->used->MAKE_B).
ASSEMBLE = (ready->assemble->used->ASSEMBLE).
||FACTORY = (MAKE_A || MAKE_B || ASSEMBLE).
composite processes

A composite process is a parallel composition of primitive processes. These composite processes can be used in the definition of further compositions.

\[ | \text{MAKERS} = (\text{MAKE}_A \parallel \text{MAKE}_B). \]

\[ | | \text{FACTORY} = (\text{MAKERS} \parallel \text{ASSEMBLE}). \]

Substituting the definition for \text{MAKERS} in \text{FACTORY} and applying the commutative and associative laws for parallel composition results in the original definition for \text{FACTORY} in terms of primitive processes.

\[ | | \text{FACTORY} = (\text{MAKE}_A \parallel \text{MAKE}_B \parallel \text{ASSEMBLE}). \]

process labeling

Two instances of a switch process:

\[ \text{SWITCH} = \text{(on->off->SWITCH)}. \]

\[ | | \text{TWO_SWITCH} = (a:SWITCH \parallel b:SWITCH). \]

An array of instances of the switch process:

\[ | | \text{SWITCHES(N=3)} = (\text{forall}[i:1..N] s[i]:SWITCH). \]

\[ | | \text{SWITCHES(N=3)} = (s[i:1..N]:SWITCH). \]

process labeling by a set of prefix labels

\[ \{a,..,a\} : P \text{ replaces every action label } n \text{ in the alphabet of } P \text{ with the labels } a_n,..,a_n. \]

Further, every transition \((n->X)\) in the definition of \(P\) is replaced with the transitions \((a_n.n,..,a_n.n)->X)\).

Process prefixing is useful for modeling shared resources:

\[ \text{RESOURCE} = (\text{acquire->release->RESOURCE}). \]

\[ \text{USER} = (\text{acquire->use->release->USER}). \]

\[ \text{RESOURCE SHARE} = (a:USER \parallel b:USER \parallel \{a,b\}:\text{RESOURCE}). \]

Process prefix labels for shared resources

Does the Resource ensure that the user that acquires the resource is the one to release it?

action relabeling

Relabeling functions are applied to processes to change the names of action labels. The general form of the relabeling function is:

\[ /\{\text{newlabel}_1/\text{oldlabel}_1,..,\text{newlabel}_n/\text{oldlabel}_n\}. \]

Relabeling to ensure that composed processes synchronize on particular actions.

\[ \text{CLIENT} = (\text{call->wait->continue->CLIENT}). \]

\[ \text{SERVER} = (\text{request->service->reply->SERVER}). \]
These “connections” hint at action relabeling. We will discuss it more formally when we capture “Structure Diagrams.”

Action relabeling - prefix labels

An alternative formulation of the client server system is described below using qualified or prefixed labels:

\[
\begin{align*}
\text{SERVERv2} &= (\text{accept.request} \rightarrow \text{service} \rightarrow \text{accept.reply} \rightarrow \text{SERVERv2}) . \\
\text{CLIENTv2} &= (\text{call.request} \rightarrow \text{call.reply} \rightarrow \text{continue} \rightarrow \text{CLIENTv2}) . \\
\text{CLIENT\_SERVERv2} &= (\text{CLIENTv2} \parallel \text{SERVERv2}) /\{\text{call/accept}\} .
\end{align*}
\]

When applied to a process P, the hiding operator \{a1..ax\} removes the action names a1..ax from the alphabet of P and makes these concealed actions "silent". These silent actions are labelled \(\tau\). Silent actions in different processes are not shared.

action hiding - abstraction to reduce complexity

Sometimes it is more convenient to specify the set of labels to be exposed:

\[
\begin{align*}
\text{USER} &= (\text{acquire} \rightarrow \text{use} \rightarrow \text{release} \rightarrow \text{USER}) \setminus \{\text{use}\} . \\
\text{USER} &= (\text{acquire} \rightarrow \text{use} \rightarrow \text{release} \rightarrow \text{USER}) \\
&\quad @\{\text{acquire}, \text{release}\} .
\end{align*}
\]

Minimization removes hidden \(\tau\) actions to produce an LTS with equivalent observable behavior:

Class Exercise

\[
\begin{align*}
\text{CLIENT} &= (\text{call} \rightarrow \text{wait} \rightarrow \text{continue} \rightarrow \text{CLIENT}) . \\
\text{SERVER} &= (\text{request} \rightarrow \text{service} \rightarrow \text{reply} \rightarrow \text{SERVER}) . \\
\text{CLIENT\_SERVERv3} &= ((\text{CLIENT} \parallel \text{SERVER}) \setminus \{\text{call/request, reply/wait}\}) \setminus \{\text{call, reply}\} .
\end{align*}
\]

Construct the LTS for CLIENT\_SERVERv3

Minimization removes hidden \(\tau\) actions to produce an LTS with equivalent observable behavior:
We use structure diagrams to capture the structure of a model expressed by the static combinators: parallel composition, relabeling and hiding.

Structure diagram for CLIENT_SERVER?

Structure diagram for CLIENT_SERVERv2?

Structure diagram for CLIENT_SERVERv3?

Structure diagram - resource sharing
Exercise

RESOURCE = (acquire->release->RESOURCE).
USER = (printer.acquire->use
->printer.release->USER).

||PRINTER_SHARE
= {a:USER||b:USER||{a,b}:printer:RESOURCE}.

1. Draw the LTS of the composed process defined above.
2. Could you avoid process re-labeling in the above definitions? Use action re-labeling instead to get an equivalent composition of two user processes and one printer process.

3.2 Multi-threaded Programs in Java

Concurrency in Java occurs when more than one thread is alive. ThreadDemo has two threads which rotate displays.

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run</td>
<td>Run</td>
</tr>
<tr>
<td>Pause</td>
<td>Pause</td>
</tr>
</tbody>
</table>

ThreadDemo Example

Two threads in the program A, B

--- (do not forget the main thread of course)

Lifecycle of thread A, B

- Running - display associated with it rotates (background = green)
- Paused - Rotation stops (background = red)

Communication:

- Thread A with main thread
- Thread B with main thread

ThreadDemo – Structure Diagram

ThreadDemo implementation in Java - class diagram

ThreadDemo creates two ThreadPanel displays when initialized.
ThreadPanel manages the display and control buttons, and delegates calls to rotate() to DisplayThread. Rotator implements the Runnable interface.
Rotator class

```java
class Rotator implements Runnable {
    public void run() {
        try {
            while (true) ThreadPanel.rotate();
        } catch (InterruptedException e) {}  
    }
}
```

Rotator implements the Runnable interface, calling ThreadPanel.rotate() to move the display. run() finishes if an exception is raised by Thread.interrupt().

ThreadPanel class

```java
public class ThreadPanel extends Panel {
    // construct display with title and segment color c
    public ThreadPanel(String title, Color c) {…}
    // rotate display of currently running thread 6 degrees
    // returns value not used in this example
    public static boolean rotate() throws InterruptedException {…}
    // create a new thread with target r and start it running
    public void start(Runnable r) {
        thread = new DisplayThread(canvas,r, …);
        thread.start();
    }
    // stop the thread using Thread.interrupt()
    public void stop() { thread.interrupt(); }
}
```

ThreadPanel manages the display and control buttons for a thread. Calls to rotate() are delegated to DisplayThread. Threads are created by the start() method, and terminated by the stop() method.

ThreadDemo class

```java
public class ThreadDemo extends Applet {
    ThreadPanel A; ThreadPanel B;
    public void init() {
        A = new ThreadPanel("Thread A",Color.blue);
        B = new ThreadPanel("Thread B",Color.blue);
        add(A); add(B);
    }
    public void start() {
        A.start(new Rotator());
        B.start(new Rotator());
    }
    public void stop() {
        A.stop();
        B.stop();
    }
}
```

ThreadDemo creates two ThreadPanel displays when initialized and two threads when started. ThreadPanel is used extensively in later demonstration programs.

Summary

- **Concepts**
  - concurrent processes and process interaction

- **Models**
  - Asynchronous (arbitrary speed) & interleaving (arbitrary order).
  - Parallel composition as a finite state process with action interleaving.
  - Process interaction by shared actions.
  - Process labeling and action relabeling and hiding.
  - Structure diagrams

- **Practice**
  - Multiple threads in Java.