Chapter 3

Concurrent Execution
Concurrent execution

**Concepts:** processes - concurrent execution and interleaving. process interaction.

**Models:** parallel composition of asynchronous processes - interleaving interaction - shared actions process labeling, and action relabeling and hiding structure diagrams

**Practice:** Multithreaded Java programs
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 \((P \parallel Q)\) represents the concurrent execution of P and Q. The operator \(\parallel\) is the parallel composition operator.

\[
\text{ITCH} = (\text{scratch} \rightarrow \text{STOP}).
\]
\[
\text{CONVERSE} = (\text{think} \rightarrow \text{talk} \rightarrow \text{STOP}).
\]
\[
\parallel \text{CONVERSE}_{-}\text{ITCH} = (\text{ITCH} \parallel \text{CONVERSE}).
\]

Possible traces as a result of action interleaving.

think talk scratch
think scratch talk
scratch think talk
parallel composition - action interleaving

ITCH
0
(0,0)

CONVERSE
0
(0,0)

CONVERSE_ITCH
0
(0,0)

2 states

2 x 3 states

from ITCH

from CONVERSE

Concurrency: concurrent execution

©Magee/Kramer
**parallel composition - algebraic laws**

**Commutative:** \((P || Q) = (Q || P)\)

**Associative:**
\[
(P || (Q || R)) = ((P || Q) || R) = (P || Q || R).
\]

Clock radio example:

\[
\text{CLOCK} = (\text{tick} -> \text{CLOCK}). \\
\text{RADIO} = (\text{on} -> \text{off} -> \text{RADIO}). \\
\text{CLOCK} || \text{RADIO} = (\text{CLOCK} || \text{RADIO}).
\]

**LTS?**  **Traces?**  **Number of states?**
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 modeled. 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.

\[
\text{MAKER} = (\text{make} \rightarrow \text{ready} \rightarrow \text{MAKER}). \\
\text{USER} = (\text{ready} \rightarrow \text{use} \rightarrow \text{USER}). \\
\text{MAKER}_\text{USER} = (\text{MAKER} \mid \mid \text{USER}).
\]

**MAKER** synchronizes with **USER** when **ready**.

**LTS?**  **Traces?**  **Number of states?**
modeling interaction - handshake

A handshake is an action acknowledged by another:

\[
\begin{align*}
\text{MAKER}_{v2} & = (\text{make} \rightarrow \text{ready} \rightarrow \text{used} \rightarrow \text{MAKER}_{v2}). \\
\text{USER}_{v2} & = (\text{ready} \rightarrow \text{use} \rightarrow \text{used} \rightarrow \text{USER}_{v2}). \\
\text{MAKER\_USER}_{v2} & = (\text{MAKER}_{v2} \mid \mid \text{USER}_{v2}).
\end{align*}
\]

Interaction constrains the overall behaviour.

Concurrency: concurrent execution
modeling interaction - multiple processes

Multi-party synchronization:

\[
\text{MAKE\_A} = (\text{makeA} \to \text{ready} \to \text{used} \to \text{MAKE\_A}).
\]

\[
\text{MAKE\_B} = (\text{makeB} \to \text{ready} \to \text{used} \to \text{MAKE\_B}).
\]

\[
\text{ASSEMBLE} = (\text{ready} \to \text{assemble} \to \text{used} \to \text{ASSEMBLE}).
\]

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

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

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

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

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.

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

*a*: P prefixes each action label in the alphabet of P with a.

Two instances of a switch process:

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

\[ \text{TWO}\_\text{SWITCH} = (a: \text{SWITCH} \mid \mid b: \text{SWITCH}) . \]

An array of instances of the switch process:

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

\[ \text{SWITCHES (N=3)} = (s[i:1..N]: \text{SWITCH}) . \]
process labeling by a set of prefix labels

{a1,...,ax}::P replaces every action label \( n \) in the alphabet of \( P \) with the labels \( a1.n,...,ax.n \). Further, every transition \( (n->X) \) in the definition of \( P \) is replaced with the transitions \( ({a1.n,...,ax.n} ->X) \).

Process prefixing is useful for modeling shared resources:

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

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

\[
|| \text{RESOURCE\_SHARE} = (a:USER || b:USER || \{a,b\}::\text{RESOURCE} ) .
\]
process prefix labels for shared resources

How does the model 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/oldlabel}_1,... \text{newlabel}_n/oldlabel}_n]\.

Relabeling to ensure that composed processes synchronize on particular actions.

CLIENT = (call->wait->continue->CLIENT).
SERVER = (request->service->reply->SERVER).
action relabeling

\[ \text{CLIENTSERVER} = (\text{CLIENT} \mid \text{SERVER}) \]

\{\text{call/request, reply/wait}\}.

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

\[
\begin{align*}
\text{SERVERv2} &= \left( \text{accept}.\text{request} \\
&\quad\quad \rightarrow \text{service} \rightarrow \text{accept}.\text{reply} \rightarrow \text{SERVERv2} \right). \\
\text{CLIENTv2} &= \left( \text{call}.\text{request} \\
&\quad\quad \rightarrow \text{call}.\text{reply} \rightarrow \text{continue} \rightarrow \text{CLIENTv2} \right). \\
\text{||CLIENT_SERVERv2} &= \left( \text{CLIENTv2} \mid \mid \text{SERVERv2} \right) \\
&\quad\quad /\{\text{call/accept}\}. 
\end{align*}
\]
action hiding - abstraction to reduce complexity

When applied to a process $P$, the hiding operator $\{a_1..a_x\}$ removes the action names $a_1..a_x$ from the alphabet of $P$ and makes these concealed actions "silent". These silent actions are labeled $\tau$. Silent actions in different processes are not shared.

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

When applied to a process $P$, the interface operator $@\{a_1..a_x\}$ hides all actions in the alphabet of $P$ not labeled in the set $a_1..a_x$. 
action hiding

The following definitions are equivalent:

\[
\text{USER} = (\text{acquire} \rightarrow \text{use} \rightarrow \text{release} \rightarrow \text{USER}) \setminus \{\text{use}\}.
\]

\[
\text{USER} = (\text{acquire} \rightarrow \text{use} \rightarrow \text{release} \rightarrow \text{USER}) \oplus \{\text{acquire, release}\}.
\]

Minimization removes hidden tau actions to produce an LTS with equivalent observable behavior.
Process $P$ with alphabet \{a,b\}.

Parallel Composition $(P \parallel Q) / \{m/a, m/b, c/d\}$

Composite process $P \parallel S = (P \parallel Q) @ \{x, y\}$
structure diagrams

We use structure diagrams to capture the structure of a model expressed by the static combinators: parallel composition, relabeling and hiding.

\[
\text{range } T = 0..3 \\
\text{BUFF} = (\text{in}[i:T] \rightarrow \text{out}[i] \rightarrow \text{BUFF}).
\]

\[||\text{TWOBUF} = ?\]
Structure diagram for CLIENT_SERVER?

CLIENT
- call
- continue
- wait

SERVER
- request
- reply
- service

Structure diagram for CLIENT_SERVERv2?

CLIENTv2
- call
- continue

SERVERv2
- accept
- service

Concurrency: concurrent execution
structure diagrams - resource sharing

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

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

Concurrency: concurrent execution
Concurrent execution  

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.
ThreadDemo model

ROTATOR = PAUSED,
PAUSED = (run→RUN | pause→PAUSED
        | interrupt→STOP),
RUN = (pause→PAUSED | {run, rotate}→RUN
       | interrupt→STOP).

THREAD_DEMO = (a:ROTATOR || b:ROTATOR)

/* stop/{a,b}.interrupt.*/

Interpret run, pause, interrupt as inputs, rotate as an output.
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.

Applet

| ThreadDemo
| init()
| start()
| stop() |

Panel

| ThreadPanel
| rotate()
| start()
| stop() |

GraphicCanvas

| Thread
| display

| DisplayThread
| rotate()

| Runnable

| Rotator
| run()

Concurrency: concurrent execution
Rotator class

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
    // return 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.