Model-Based Concurrency, Part Two

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 5828 — Lecture 13 — 02/23/2010

© University of Colorado, 2010
Credit Where Credit is Due

The remainder of the slides in this presentation were created by Magee and Kramer for Chapter 3 of the Concurrency textbook.
Chapter 3

Concurrent Execution

Concurrency: concurrent execution
Concurrency: 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 \((P \mid\mid Q)\) represents the concurrent execution of P and Q. The operator \(\mid\mid\) is the parallel composition operator.

\[
\text{ITCH} = (\text{scratch} \rightarrow \text{STOP}).
\]
\[
\text{CONVERSE} = (\text{think} \rightarrow \text{talk} \rightarrow \text{STOP}).
\]
\[
\mid\mid\text{CONVERSE_ITCH} = (\text{ITCH} \mid\mid \text{CONVERSE}).
\]

Possible traces as a result of action interleaving.

think \(\rightarrow\) talk \(\rightarrow\) scratch
think \(\rightarrow\) scratch \(\rightarrow\) talk
scratch \(\rightarrow\) think \(\rightarrow\) talk

Disjoint alphabets
parallel composition - action interleaving

ITCH

CONVERSE

The combined process is the Cartesian product of ITCH and CONVERSE

CONVERSEITCH

The combined process is the Cartesian product of ITCH and CONVERSE

Concurrency: concurrent execution

2 states

3 states

2 x 3 states
parallel composition - algebraic laws

Commutative: \((P \parallel Q) = (Q \parallel P)\)
Associative: \((P \parallel (Q \parallel R)) = ((P \parallel Q) \parallel R) = (P \parallel Q \parallel R)\).

Clock radio example:

\[
\begin{align*}
\text{CLOCK} &= (\text{tick} \rightarrow \text{CLOCK}). \\
\text{RADIO} &= (\text{on} \rightarrow \text{off} \rightarrow \text{RADIO}). \\
\text{||CLOCK\_RADIO} &= (\text{CLOCK} \mid \mid \text{RADIO}).
\end{align*}
\]

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.

**MAKER** = (make->ready->MAKER).
**USER** = (ready->use->USER).

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

| |MAKER_USER = (MAKER || USER).

**LTS?** *Traces?*  **Number of states?**

Concurrency: concurrent execution

Non-disjoint alphabets
shared alphabets affect the Cartesian product

◆ BILL = (play → meet → STOP).
◆ BEN = (work → meet → STOP).

◆ Each process has three states (initial, after first action, after second action)
◆ Cartesian product should produce 9 states (3 x 3)

◆ But LTS contains only 5 states! Why?
◆ Due to rules governing shared actions

◆ Full Cartesian Product: (initial, initial), (initial, work), (initial, meet), (play, initial), (play, work), (play, meet), (meet, initial), (meet, work), (meet, meet)

◆ But due to rules governing shared actions, the red tuples are not permitted; Both processes must be ready to transition to the state after the “meet” action at the same time
modeling interaction - handshake

A handshake is an action acknowledged by another:

\[
\text{MAKER}_v2 = (\text{make} \rightarrow \text{ready} \rightarrow \text{used} \rightarrow \text{MAKER}_v2).
\]
\[
\text{USER}_v2 = (\text{ready} \rightarrow \text{use} \rightarrow \text{used} \rightarrow \text{USER}_v2).
\]
\[
\text{MAKER\_USER}_v2 = (\text{MAKER}_v2 \mid \mid \text{USER}_v2).
\]

Interaction constrains the overall behaviour.
modeling interaction - multiple processes

Multi-party synchronization:

\[
\begin{align*}
\text{MAKE}_A & = (\text{makeA} \rightarrow \text{ready} \rightarrow \text{used} \rightarrow \text{MAKE}_A). \\
\text{MAKE}_B & = (\text{makeB} \rightarrow \text{ready} \rightarrow \text{used} \rightarrow \text{MAKE}_B). \\
\text{ASSEMBLE} & = (\text{ready} \rightarrow \text{assemble} \rightarrow \text{used} \rightarrow \text{ASSEMBLE}). \\
\text{FACTORY} &= (\text{MAKE}_A \mid \mid \text{MAKE}_B \mid \mid \text{ASSEMBLE}).
\end{align*}
\]
A composite process is a parallel composition of primitive processes. These composite processes can be used in the definition of further compositions.

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

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

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

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

\[ a: P \text{ prefixes each action label in the alphabet of } P \text{ with } a. \]

Two instances of a switch process:

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

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

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 prefixing is useful for modeling shared resources:

\[ \text{RESOURCE} = (\text{acquire} -> \text{release} -> \text{RESOURCE}) . \]
\[ \text{USER} = (\text{acquire} -> \text{use} -> \text{release} -> \text{USER}) . \]
\[ | | \text{RESOURCE}_\text{SHARE} = (a : \text{USER} | | b : \text{USER} | | \{a, b\} :: \text{RESOURCE}) . \]
process prefix labels for shared resources

How does the model ensure that the user that acquires the resource is the one to release it?
Relabeling functions are applied to processes to change the names of action labels. The general form of the relabeling function is:

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

Relabeling to ensure that composed processes synchronize on particular actions.

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

Note that both newlabel and oldlabel can be sets of labels.
action relabeling

\[ ||\text{CLIENT\_SERVER} = (\text{CLIENT} || \text{SERVER}) \]

\[ /\{\text{call/request, reply/wait}\}. \]
action relabeling - prefix labels

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

\[
\text{SERVERv2} = (\text{accept.request} \\
\quad \rightarrow \text{service} \rightarrow \text{accept.reply} \rightarrow \text{SERVERv2}).
\]

\[
\text{CLIENTv2} = (\text{call.request} \\
\quad \rightarrow \text{call.reply} \rightarrow \text{continue} \rightarrow \text{CLIENTv2}).
\]

\[
\text{||CLIENT\_SERVERv2} = (\text{CLIENTv2} \mid \mid \text{SERVERv2}) \\
\quad \{/\text{call/accept}\}.
\]
action hiding - abstraction to reduce complexity

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 labeled \textit{tau}. Silent actions in different processes are not shared.

Sometimes it is more convenient to specify the set of labels to be \textit{exposed}....

When applied to a process P, the interface operator @\{a1..ax\} hides all actions in the alphabet of P not labeled in the set a1..ax.
**action hiding**

The following definitions are equivalent:

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

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

Minimization removes hidden \(\tau\) actions to produce an LTS with equivalent observable behavior.
structure diagrams – systems as interacting processes

Process P with alphabet \{a,b\}.

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

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

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

range $T = 0..3$

$BUFF = (in[i:T]->out[i]->BUFF)$.

$||TWOBUFF = (a:BUFF || b:BUFF) / \{in/ a.in, a.out/b.in, out/b.out\} @ \{in, out\}$.

---

Concurrency: concurrent execution
structure diagrams

Structure diagram for CLIENT_SERVER:

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

||CLIENT_SERVER = (CLIENT || SERVER)

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

Concurrency: concurrent execution

©Magee/Kramer 2nd Edition
structure diagrams - resource sharing

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

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

Concurrency: 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

\[
\begin{aligned}
\text{ROTATOR} &= \text{PAUSED}, \\
\text{PAUSED} &= (\text{run} \rightarrow \text{RUN} \mid \text{pause} \rightarrow \text{PAUSED} \\
&\quad \mid \text{interrupt} \rightarrow \text{STOP}), \\
\text{RUN} &= (\text{pause} \rightarrow \text{PAUSED} \mid \{\text{run, rotate}\} \rightarrow \text{RUN} \\
&\quad \mid \text{interrupt} \rightarrow \text{STOP}).
\end{aligned}
\]

\[
\begin{aligned}
||\text{THREAD}\_\text{DEMO} &= (\text{a:ROTATOR} \mid \mid \text{b:ROTATOR}) \\
\text{/}{\text{stop/\{a,b\}.interrupt}}.
\end{aligned}
\]
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
    // 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.