Concurrent Programming 19530-V (WS01)

Lecture 12: Modeling Dynamic Systems & Process Communication

Dr. Richard S. Hall
rickhall@inf.fu-berlin.de

Part 1: Modeling Dynamic Systems

In all of the models we have discussed and implemented so far, threads are created during initialization and they run until program termination (for the most part). What about modeling programs where the number of active threads varies during execution?

Dynamic Thread Example

Golf Course Example

A golf course has a limited number of golf balls available for its players. Players check out golf balls to play golf and return them after they are finished. Expert players, who never lose balls, only take one or two balls. Novice players take more balls so that they have extras in case any get lost. All players must return the same number of balls they checked out, so they must buy replacement balls if any are lost.

In this example, new players arrive dynamically.

This is an example of a resource allocation problem.

Modeling Dynamic Systems

- What is the perceived difficulty with using FSP / LTS for dynamic systems?
  - These techniques require a static number of processes in order to permit analysis
  - Dynamic systems do not have a static number of processes
- We will start by taking a quick look at the implementation and model of the ***Golf Course***

Golf Course Implementation

Players threads and their desired ball requests are created by pushing the button corresponding button. Player names are generated using an ordered letter from the alphabet concatenated with the number of balls required. New player threads temporarily appear in the "new" box and finishing player threads temporarily appear in the "end" box.

Golf Course Implementation

```
public interface Allocator {
   public void get(int n)
       throws InterruptedException;
   public void put(int n);
}

public class SimpleAllocator implements Allocator {
   private int available;
   public SimpleAllocator(int n) { available = n; }
   public synchronized void get(int n)
      throws InterruptedException {
      while (n > available) wait();
      available -= n;
   }
   public synchronized void put(int n) {
      available += n;
      notifyAll();
   }
}
```

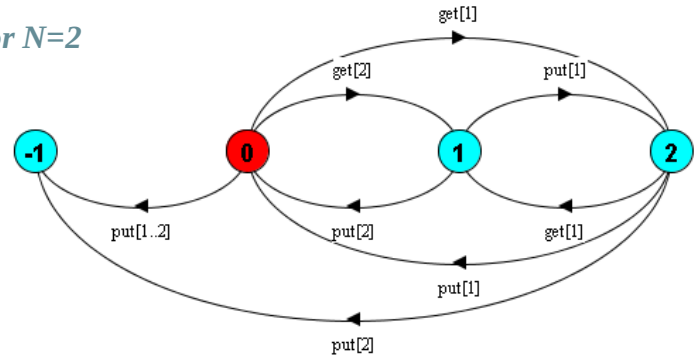
Golf Course Implementation

```
public class Player implements Runnable {
   private GolfCourse gc;
   private String name;
   private int nballs;
   public Player(GolfCourse g, int n, String s) {
      gc = g; nballs = s; name = s;
   }
   public void run() {
      try {
         gc.getGolfBalls(nballs, name);
         Thread.sleep(gc.playTime);
         gc.relGolfBalls(nballs, name);
      } catch (InterruptedException ex) { }
   }
}
...
Thread t = new Thread(new Player());
t.start();
```

Golf Course Model

```
const N=5    //maximum number of golf balls
range B=0..N //available range

ALLOCATOR = BALL[N],
BALL[b:B] = (when (b>0) get[i:1..b]->BALL[b-i]
            |put[j:1..N]->BALL[b+j]).
```

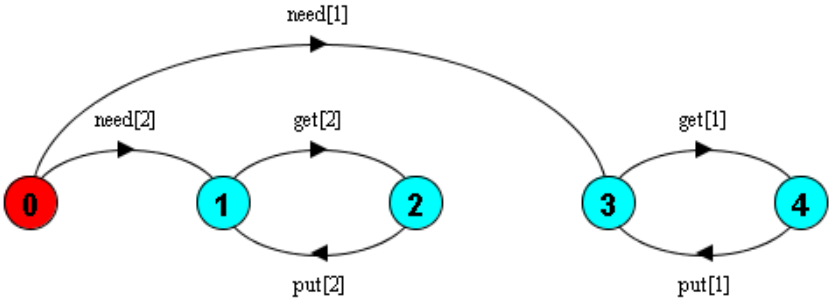
LTS for N=2



Golf Course Model

```
range R=1..N //request range

PLAYER      = (need[b:R]->PLAYER[b]),
PLAYER[b:R] = (get[b]->put[b]->PLAYER[b]).
```

LTS for N=2



Golf Course Model

- How do we model the potentially infinite stream of dynamically created players?
  - We can't with FSP / LTS is the short answer, because we cannot model infinite spaces
  - However, we can model infinite behaviors that are repetitive
    - We do not need to model that each player is distinct
    - We model a fixed population of players who continuously repeat the actions of playing

```
set Experts = {alice, bob, chris}
set Novices = {dave, eve}
set Players = {Experts, Novices}
```

Golf Course Model

```
HANDICAP =
  ({Novices.{need[3..N]},Experts.need[1..2]}
      ->HANDICAP)
  +{Players.need[R]}.

||GOLFCOURSE = (Players:PLAYER
                ||Players::ALLOCATOR
                ||HANDICAP).
```

HANDICAP distinguishes between novices and experts. Alphabet extension is used to ensure that when **HANDICAP** is composed with **PLAYER** processes they are inhibited from performing any other need actions.

Part 2: Process Communication

In previous examples we have used processes interacting through shared variables to illustrate concurrency issues. Shared variables are simply one form of process communication.

Now we examine alternative approaches to process communication.

Message Passing

- Instead of using shared variables, processes can communicate by sending and receiving messages
  - Conceptually this means that the processes do not share memory, but still reside on the same computer
  - A result of not sharing memory is that processes may reside on different computers connected via a network
  - Since most distributed systems use some form of message passing mechanism, it is easy to see that concurrency is intimately tied to distributed systems

Message Passing

- Message passing primitive operations
  - ***send*** - send a message to someone else
  - ***receive*** - receive a message from someone else
- Two basic models for ***send*** / ***receive*** primitive
  - ***Synchronous*** = writing/reading a message blocks until someone reads/writes
  - ***Asynchronous*** = writing/reading does not block, written messages are buffered and if no message is available the read operation returns immediately
- These message primitives are one-way
  - Messages are transmitted from the sender to the receiver

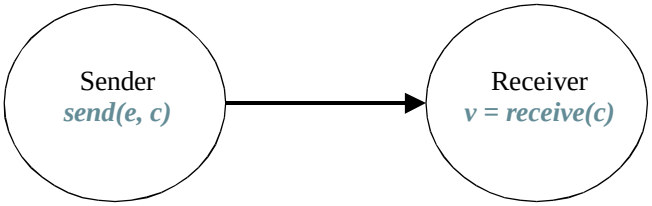
Message Passing

- How are messages addressed?
  - Addressed directly to the destination process
  - Addressed indirectly to some intermediate entity
- One model of message passing using the notion of a ***channel***
  - Messages are sent to and received from a channel
  - A channel connects one sender and one receiver, thus communication is ***one-to-one***

Synchronous Channel Message Passing

send(e, c) - send the value of the expression ***e*** to channel ***c***. The process calling send is blocked until the message is received by another process.

v = receive(c) - receive a value into local variable ***v*** from a channel ***c***. The calling process is blocked until a message is sent to the channel.



Selective Message Receive

With blocking semantics it is inconvenient receiving from multiple channels, therefore some message passing systems implementing some form of selective receive; the general form of a select statement is (this is not FSP):

```
select
  when G1 and v1 = receive(chan1) => S1;
  or when G2 and v2 = receive(chan2) => S2;
  or when GN and vN = receive(chanN) => SN;
end
```

G_i is a boolean guard which indicates that a ***receive*** is eligible if the guard is ***true***; the ***select*** statement chooses an eligible ***receive*** for which there is a sender waiting to ***send***.

Selective Message Receive

Some message passage implementations allow you to do a polling receive (this is not FSP):

```
select
    v = receive(chan) => S;
  else
    S_ELSE;
end
```

If there is no sender ready to ***send*** on *chan* then the ***else*** part is chosen for execution.

Modeling Synchronous Messaging

Modeling the sender and receiver is straight-forward...

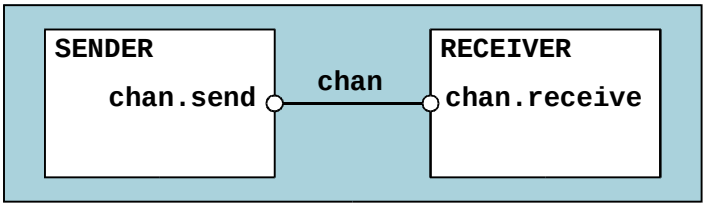
```
range M = 0..9
SENDER = SENDER[0],
SENDER[e:M] = (chan.send[e]
                  ->SENDER[(e+1)%10]).
RECEIVER = (chan.receive[v:M]->RECEIVER).
```

How do we model the channel entity?

Modeling Synchronous Messaging

Since sending and receiving are ***synchronous***, we know that this means that they share an action, in this case, they are the same action. Therefore combining a sender and a receiver is a matter of renaming.

```
||SynchMsg = (SENDER || RECEIVER)
             /{chan/chan.{send, receive}}.
```



Can we avoid renaming?

Modeling Synchronous Messaging

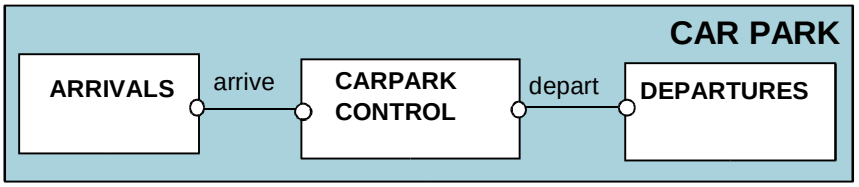
To avoid relabeling, we can model the send action directly as `chan[e]` and the receive action as `chan[v:M]`. The only difference is that a ***receive*** is modeled as a choice between a set of `M` values, whereas a ***send*** specifies a specific value `e`.

Message Operation	FSP Model
send(*e*, *chan*)	`chan[e]`
v = **receive**(*chan*)	`chan[e:M]`

Modeling Selective Message Receive

Recall the car park example:

```
CARPARKCONTROL(N=4) = SPACES[N],
SPACES[i:0..N] = (when(i>0) arrive->SPACES[i-1]
                 |when(i<N) depart->SPACES[i+1]).

ARRIVALS   = (arrive->ARRIVALS).
DEPARTURES = (depart->DEPARTURES).

||CARPARK = (ARRIVALS||CARPARKCONTROL(4)||DEPARTURES).
```



Modeling Selective Message Receive

- For the car park, we do not need to change the model for message passing
  - The when clauses directly map to receiver selection
  - The `ARRIVALS` and `DEPARTURES` processes are considered to be sending messages on channels for signaling arrivals and departures, respectively
- In the original implementation, we implemented the `CARPARKCONTROL` process as a monitor
  - For message passing, we implement it as a thread that receives messages from `ARRIVALS` and `DEPARTURES` to indicate when a car has arrived or departed, respectively
- The implementation of `ARRIVALS` and `DEPARTURES` are identical, they just send messages to channels instead of invoking a method

Modeling Asynchronous Messaging

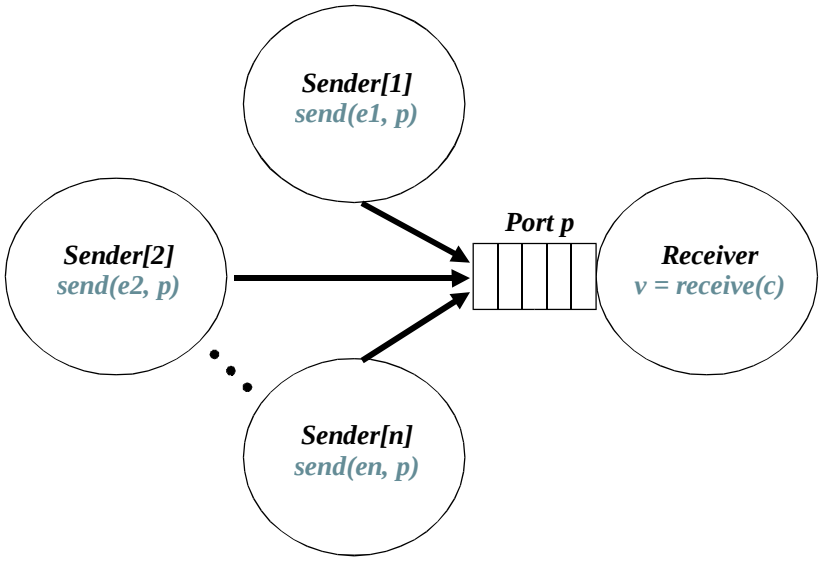
- In asynchronous message passing the send operation does not block, while the receive operation typically blocks until a message is available
- A ***port*** is an example of an asynchronous message passing concept
  - Messages are held in a queue until received
  - A port may have many senders, but only one receiver, thus communication is ***many-to-one***

Port Operations

send(e, p) - send the value of the expression ***e*** to port ***p***. The process calling send is not blocked. The message is queued at the port if a receiver is not waiting.

v = receive(p) - receive a value into local variable ***v*** from a port ***c***. The calling process is blocked if there are no messages in the message queue.

Conceptual Illustration of a Port



Modeling Asynchronous Messaging

- Asynchronous message passing model not as simple as synchronous model, some of the difficulties are
  - Message queues associated with a port are potentially unbounded, we know this is a problem for FSP
    - We must adopt the same approach we used for semaphores and model the queue as finite and allow an overflow error
  - Given the range of messages and the size of the message queue, it is very easy for the state space of a port model to explode

Modeling Asynchronous Messaging

Assume that we are again using an integer value between 0 and 9...

```
range M = 0..9
set S = {[M],[M][M]}

PORT                    // empty state
   = (send[x:M]->PORT[x]),
PORT[h:M]               // one message queued
   = (send[x:M]->PORT[x][h]
     |receive[h]->PORT),
PORT[t:S][h:M]          // two or more messages queued
   = (send[x:M]->PORT[x][t][h]
     |receive[h]->PORT[t]).
```

The set **S** defines the set of values that can be taken by the tail of the queue when the queue contains two or more messages; essentially it is a set of types for the **PORT** parameters.

Modeling Asynchronous Messaging

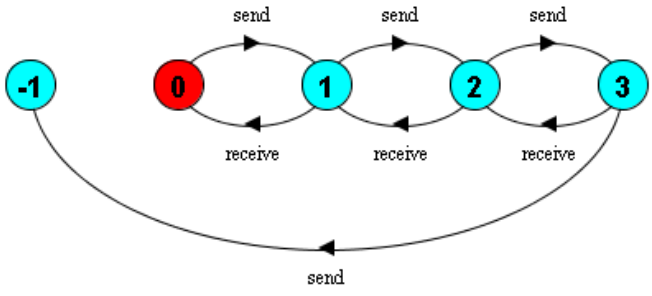
- To model a port that can queue up to four messages, we must change **S** to
  - `set S = {[M],[M][M],[M][M][M][M]}`
- With a queue for three messages, a LTS with 1111 states is generated; with four messages is a total of 11111 states
- Clearly these graphs are too big to view
- We can abstract away the value of the messages to examine the send and receive actions

Modeling Asynchronous Messaging

```
||ABSTRACTPORT = PORT
          /{send/send[M],receive/receive[M]}.
```

This process simply ignores the value of the individual messages, treating all sends and receives as the same action.

This results in a simplified LTS with observable behavior, for `range M = 0..3`:



Modeling Asynchronous Messaging

```
SENDER = SENDER[0],
SENDER[e:M] = (port.send[e]->SENDER[(e+1)%10]).

RECEIVER = (port.receive[v:M]->RECEIVER).

||AsynchMsg = (s[1..2]:SENDER || port:PORT || RECEIVER)
              /{s[1..2].port.send/port.send}.
```

Analysis? It is possible to overflow the buffer...

```
Trace to property violation in port:PORT:
       s.1.port.send.0
       s.1.port.send.1
       s.1.port.send.2
       s.1.port.send.3
```

Rendezvous Message Passing

- In ***rendezvous*** message passing, a client sends a request message to a server
  - Also referred to as ***request-reply*** message passing
- Requests messages are queued on an ***entry*** in FIFO order
  - Requests are ***many-to-one***
- The server accepts requests from the entry and sends a reply message to client on completion
- The client blocks waiting for the reply message
  - Replies are ***one-to-one***

Rendezvous Message Passing

res = call(e, req) - send the value ***req*** as a request message which is queued to the entry ***e***. The calling process is blocked until a reply message is received into the local variable ***res***.

req = accept(e) - receive the value of the request message from the entry ***e*** into the local variable ***req***. If there are no request messages queued to the entry, then the server process is blocked.

reply(e, res) - send the value ***res*** as a reply message to entry ***e***.

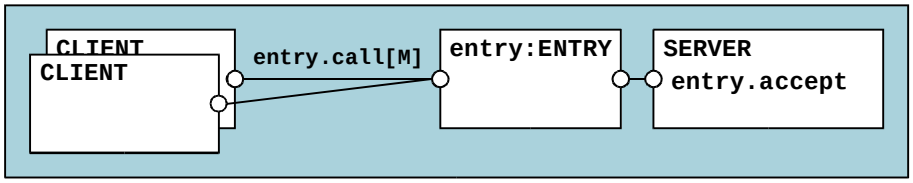
Called "rendezvous" since the client and server processes meet and synchronize when the server performs a service for the client.

Modeling Rendezvous Messaging

```
// Reply channels
set M = {replyA, replyB}

// Create an entry from a port
||ENTRY = PORT /{call/send, accept/receive}.

// Server receives on port, replies on channel
SERVER = (entry.accept[ch:M]->[ch]->SERVER).

// Client requests on port, receives on channel
CLIENT(CH='reply) = (entry.call[CH]->[CH]->CLIENT).
```

Modeling Rendezvous Messaging

```
||EntryDemo = (CLIENT('replyA) || CLIENT('replyB)
              || entry:ENTRY || SERVER).
```



Notice that we did not prefix the client processes like we normally do; this is because the parameter causes them to have distinct alphabets. For example, the alphabet for **`CLIENT('replyA)`** is:

`{entry.call.replyA, replyA}`.

Modeling Rendezvous Messaging

From the viewpoint of a client, other than syntactic differences, a call on an **`entry`** is very similar to calling a monitor access method. Only one client request at a time can be serviced, thus mutual exclusion is guaranteed.

What is the only conceptual difference between the two?

The client thread is used to handle the request in monitor methods, whereas the server thread is used to handle the request in an entry.

Java Detour: Message Passing

- Sockets
  - A ***socket*** is an inter-process message passing mechanism; similar to a ***port*** as we have defined it
  - You can use sockets to create concurrency between separate processes on separate machines
  - A socket address is an IP address and a port number
    - This port number has nothing to do with a ***port***, it is merely a number to uniquely distinguish sockets on a particular machine
  - The protocol for communication over a socket is typically TCP, UDP, or RDP

Java Detour: Message Passing

- Connection-oriented socket
  - A stateful connection or stream (e.g., telephone)
  - Offers certain guarantees
    - Arrival, order
  - Local sockets (client-side) can ***connect*** to a remote socket
  - You ***listen*** for connections to a socket (server-side)
  - When a connection is detected you can ***accept*** the connection (server-side)
  - In Unix, to deal with many connection requests you can ***select*** them (server-side, but not in Java < 1.4)
  - Transferring data via ordinary ***read/write*** commands

Java Detour: Message Passing

- Datagram socket
  - Communicate by sending self-contained packets (e.g., mailing a letter)
  - Does not offer guarantees
  - Arrival, order
  - You ***create*** a datagram socket
  - Simply ***receive*** or ***send*** data

Java Detour: Message Passing

```
public class Server {
  public Server() {
    try {
      ServerSocket s = new ServerSocket(1968);
      while (true)
      {
        Socket cs = s.accept();
        serviceClient(cs);
      }
    } catch (IOException ex) {
      System.err.println("Server: " + ex);
      System.exit(-1);
    }
  }

  public void serviceClient(Socket cs) {
    try {
      ObjectInputStream ois = new ObjectInputStream(cs.getInputStream());
      System.out.println(ois.readUTF());
      ObjectOutputStream oos = new ObjectOutputStream(cs.getOutputStream());
      oos.writeUTF("Tschuess");
      oos.flush();
      cs.close();
    } catch (IOException ex) {
      System.err.println("Server: " + ex);
    }
  }
}
```

Java Detour: Message Passing

```
public class Client {
  public Client() {
    try {
      Socket s = new Socket("heavy.inf.fu-berlin.de", 1968);
      ObjectOutputStream oos = new ObjectOutputStream(s.getOutputStream());
      oos.writeUTF("Hallo");
      oos.flush();
      ObjectInputStream ois = new ObjectInputStream(s.getInputStream());
      System.out.println(ois.readUTF());
      s.close();
    } catch (IOException ex) {
      System.err.println("Client: " + ex);
      System.exit(-1);
    }
  }
}
```

You will learn all about this stuff in distributed systems...