{a:"s" b:"t" 1:"a"}

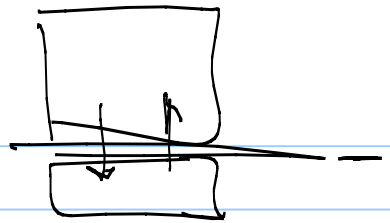
Lecture 9

Note Title 2/7/2008

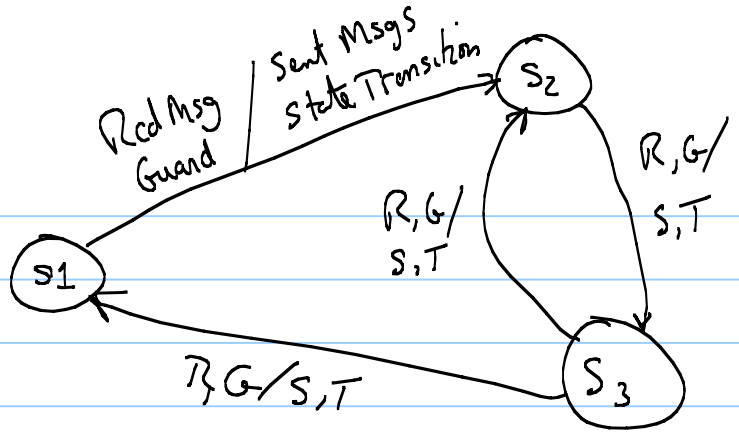
Fig 5.14

```
proc {MsgLoop S1 Procs}
   case S1
      of add(I Proc Sync)|S2 then Procs2 in
         Procs2={AdjoinAt Procs I Proc}
         Sync=unit
         {MsgLoop S2 Procs2}
   [] msg(I M)|S2 then
         try {Procs.I M} catch _ then skip end
         {MsgLoop S2 Procs}
   [] ...
```

Designing with Concurrency 5.4



- Interface for each Agent —
  what kinds of messages does it send and receive
  — what order do messages occur in — semantic specification
- State - what set of values are needed to figure out what messages to send - how do we represent the state - record, list,



Guard – Boolean Exp n State

state: discrete set of nodes along with a set of variables that can be updated.

UML – State Charts

Erlang*

Ericsson Telecom

high Availability, on-the-fly updates

originally proprietary — open source now

OTP — Open Telecom Platform

Mnesia — distributed / replicated database

Strict functional core:

before calling a function the arguments are evaluated

single-assignment variables

pattern matching on arguments

Dynamic strong typing.

Concurreny is fundamental:
Transparent Distribution
Module system – separate compilation and code replacement
Watch-dog failure detection and reporting mech.

Variable names start Upper Case } like Oz
atoms are all lower case
function names are lower case.

Oz-style
rectangle(X, Y)

function definition

tuple

atom

area/1

```
area({rectangle, X, Y}) -> X * Y;
area({circle, Radius}) -> 3.14 * Radius * Radius;
area(rectangle, X, Y) ->
```

area/3

[,]

Threads ~ processes

function

module

function

spawn/1

spawn/3

```
Pid = spawn(F)
Pid = spawn(M, F, A)
```

argument list

Each has a mailbox – unbounded queue of messages kept in FIFO order

- A process can selectively receive from its mailbox

Pid ! value — asynchronous buffered send.

receive
    Pat1 [when G1] -> Body1;
    ..
    PatN [when G2] -> BodyN;
    [after Expr -> BodyT]
end

after 0 – poll

after infinity

A variable bound by every pattern is visible after the receive.

Simple RPC -

Server

```
-module (areaserver).
-export ([start/0, loop/0]).
start() -> spawn(areaserver, loop, []);
loop() -> receive
              {From, Shape} -> From ! area(Shape), loop()
end.
```

{self(), area(Shape)}

Client

```
Pid = areaserver:start(),
    Pid ! {self(), {rectangle, 3, 4}}, receive Ans -> ... end, ...
```

encapsulate the rpc pattern

```
rpc (Pid, Request) ->
    Pid ! {self(), Request},
    receive
        Ans -> Ans
    end.
```

rpc (Pid, {rectangle, 3, 4})

```
rpc (Pid, Request) ->
    Pid ! {self(), Request},
    receive {Pid, Ans} -> Ans
    end.
```