Lecture 17
Concurrent Programming

2nd November 2003

CCS Continued

Example agents:

\[
\begin{align*}
C_1 & \equiv \text{in}.\text{out}.0 \\
C_2 & \equiv \text{in}.\text{out}.C_2 \\
D_1 & \equiv \text{in}.\text{outa}.\text{outb}.D_1 \\
D_2 & \equiv \text{in}.(\text{outa}.0 + \text{outb}.0) \\
D_3 & \equiv \text{in}.(\text{outa}.D_3 + \text{outb}.D_3) \\
D_4 & \equiv \text{in}.\text{outa}.D_4 + \text{in}.\text{outb}.D_4
\end{align*}
\]

Understand how each of these agents evolves as communication occurs.

Note well the difference in behavior between \(D_3\) and \(D_4\). The former receives a message on \textit{in} then offers a choice to the environment of either \textit{outa} or \textit{outb}. The latter, after accepting a message from \textit{in} will be prepared to send only one of \textit{outa} or \textit{outb}. \(D_3\) is like:

\[
\begin{align*}
\text{recv(in);} \\
\text{select}([\text{outa, outb}])
\end{align*}
\]

while \(D_4\) is like:

\[
\begin{align*}
\text{choice} = \text{random boolean value;} \\
\text{if choice} \{ \\
\text{recv(in);} \; \text{send(outa);} \\
\text{else} \{ \\
\text{recv(in);} \; \text{send(outb);} \\
\}
\end{align*}
\]

The important point here: \(D_3\) leaves the choice of whether \textit{outa} or \textit{outb} happens next to the environment (other concurrent processes), while \(D_4\) makes an \textit{internal choice}.  

1
Evolution of concurrent agents

If two agents composed in parallel offer complementary communications, both can evolve to their next state:

\[
E_1 \equiv in'.m.E_1 \\
E_2 \equiv m'.out.E_2 \\
Src \equiv in.0 \\
Sink \equiv out'.0 \\
System_1 \equiv E_1 | E_2 \\
System_2 \equiv System_1 | Src \\
System_3 \equiv System_1 | Src | Sink
\]

Describe the evolution of System_1 in isolation; describe the evolution of System_2 and System_3.

Linda mechanisms

Linda not a language but rather a conceptual approach to concurrency that relies on the notion of a tuple space: a shared collection of labelled data records. Example tuples:

- ("chan1", 1, 2, 3)
- ("chan1", 'a', 'b', 'c')
- ("cat", "grey", "5-toed")
- ("cat", "orange", "3-legged")

One way to think of tuple space is as an unordered communication channel.

Operations on tuple space: OUT, IN, RD, INP, RDP, EVAL. OUT places a new tuple in the tuple space:

\[
\text{OUT} \ (\text{tag}, \text{val1}, \text{val2}, ..., \text{valn})
\]

IN removes a tuple from tuple space using matching and assignment:

\[
\text{IN} \ (\text{tag}, \text{var-or-val1}, ..., \text{var-or-valn})
\]

A tuple matches an IN operation if the tags are the same and the types of the fields of the tuple are the same as the types of the fields in the argument to IN and the argument to IN have the same values in the positions that the argument to IN has values. Some tuple that matches the IN is removed from the tuple store and the values in the tuple corresponding to the variables in the argument to IN are assigned to those variables. If no such tuple exists, the IN operation blocks.

RD is similar to IN but does not remove the matching tuple.

RDP and INP are similar to RD and IN but do not block if there is no matching tuple.

Finally, EVAL is like OUT, but initiates concurrent evaluation of the tuple’s fields:
EVAL (tag, f1(args1), f2(args2), ...fn(argsn))

A concurrent process will be started to evaluate the calls to f1, f2, ... fn. When it is finished the resulting tuple is placed in the tuple store.

**Barrier Example**

Suppose we need a barrier requiring 4 processes to arrive before any proceed further; We can initialize it with

```plaintext
OUT("barrier", 0)
```

Each time a process arrives at the barrier it executes

```plaintext
IN("barrier", ?counter);
OUT("barrier", counter+1);
RD("barrier", 4);
```

Question: why is there not a race condition in reading and updating counter in the first two lines?

Question: how could the barrier be reused?

**Motivation for Linda**

- *communication orthogonality*: neither senders nor receivers identify the other party to a communication (compare with channels)

- *space uncoupling*: when implemented in a distributed setting tuples deposited by any process at any location may be received by any process at any other location

- *time uncoupling*: a tuple added to the tuple-space will remain there until it is removed by IN. In some implementations this can extend to persistent tuple stores that last beyond the execution of all programs that use them

- *distributed sharing*: as seen above in the barrier example, Linda tuples can be used to simulate shared variables even in a setting where there is no shared address space.