CS 6112 (Fall 2011)

Foundations of Concurrency

29 September 2011

Scribe: Carlos Otero

Cornell University Department of Computer Science

1 π-Calculus Review

$$\begin{aligned} P &:: M \mid P_1|P_2 \mid \nu x\ P \mid !P \\ M &::= 0 \mid \pi.P \mid M_1 + M_2 \\ \pi &:= \tau \mid \overline{x}\left\langle y \right\rangle \mid [x=y]\pi \end{aligned}$$

2 Introduction

This session we will talk about Asynchronous π-Calculus.

Asynchronous π-Calculus accurately models real time systems that have some characteristics like:

- There exists delay on channels
- There are buffers on channels
- Information might be dropped if buffers overflow

Besides its practical implications, Asynchronous π-Calculus present interesting theoretical challenges. All the equivalent relationships we built for CCS become more subtle once we incorporate the model of Asynchronous π-Calculus.

In full π-Calculus, synchronization occurs on guards since residual processes fire whenever a guard evaluates *true*, in Asynchronous π-Calculus, we would like to avoid unnecessary synchrononization happening because of the guards. We would like synchronization to only happen whenever *send* and *receive* operations happen.

For the rest of the session we will:

- Define Asynchronous π-Calculus
- Examples how Asynchronous π-Calculus is used
- How to convert π-Calculus into Asynchronous π-Calculus

3 Definition. Asynchronous π-Calculus

This method is described by Kohei Honda [3].

$$\begin{aligned} P &::= \overline{x}y \mid M \mid P_1|P_2 \mid \nu x\ P \mid !P \\ M &::= 0 \mid x(y).P \mid \tau.P \mid M_1 + M_2 \end{aligned}$$

Note: The *send* notation used by Kohei Honda does not use $\langle . \rangle$ for *sends*.
Sends are banned to guard processes, in order to avoid unnecessary synchronization. *Receives* can guard processes, and *sends* can only guard the null process. By using this method whenever we attempt to send, it immediately commits. If one would like to do sequencing, then an explicity acknowledge channel is required to send back the value that was sent.

A *send* is waiting to react with a symmetric *receive* action. Sequencing can still happening by using private channels to acknowledge a channel. to perform *send* and *receive* actions.

Summations are also restricted so that only *sends* and τ guarded processes can appear. Hence it is impossible to build a process that looks like:

$$\overline{x}y + w(x).P$$

The reaction rule for this process is:

$$\frac{}{\overline{x}\,\langle y \rangle \;|\; (x(z).P + M) \rightarrow P\{y/z\}} : React$$

3.1 Lossy channels

This model correctly handles delay and buffering on the channel, however it still considers lossless channels.

To allow lossy channels, once could explicitly model *loss* usign a loss process:

$$lserver \equiv !x(y).0$$

One can make the analogy of this process to the the /dev/null device in any Unix-like operating system.

3.2 Order and sequencing

We can enforce ordering by intentionally introducing data dependencies. The following example shows how we can enforce particular ordering of events:

$$\begin{aligned} P \equiv \nu x,\ w \qquad & (\overline{x}w \;|\; w(u).(\overline{u}a \,|\, w(z).\overline{z}b) \;| \\ & x(w).\nu u(\overline{w}u \;|\; u(y).\nu\, z(\overline{w}z \,|\, z(s).Q))) \end{aligned}$$

In this example, the sequence of events that happen are:

- w is sent on channel x
- u is sent on channel w
- a is sent on channel u and it is stored in y
- z is sent on channel w
- b is sent on channel z and it is stored in s

At the end, we have a residual process $Q\{a/y, b/s\}$. This process shows how sequencing was achieved between channels $x \rightarrow y \rightarrow z$ using private channels w and u for sequencing.

4 π-Calculus translation into Asynchronous π-Calculus

This section describes a way to integrate and encode any process of the π-Calculus into asynchronous π-Calculus.
$[\![]\!]$:Operator that defines the equivalence between full π-Calculus into Asynchronous π-Calculus.

4.1 Translation of ***send*** and ***receive*** actions

First, we need to change the protocols for *sends* and *receives*:

$$\begin{aligned}[\![\overline{x}\langle y\rangle .P]\!] &\triangleq \nu\, w(\overline{x}w \mid w(u).(\overline{u}\langle y\rangle \mid [\![P]\!]))\\ [\![x(z).Q]\!] &\triangleq x(w).\nu\, u(\overline{w}u \mid u\langle z\rangle .[\![Q]\!])\end{aligned}$$

The translation above will *send* a channel on x. The symmetric *receive* action happens. Afterwards, the data is sent along with that channel in parallel with the execution of the residual process P.

4.2 Translation of sums

We need to split a sum comprising *send*, *receive* and τ actions into an equivalent program that includes only sums of *send* <u>OR</u> *receives*.
We will first extend the encoding of asynchronous π-Calculus:

$$\begin{aligned}P &::= \overline{x}y \mid M \mid P_1|P_2 \mid \nu xP \mid !P \mid N\\ M &::= \sum_i x_i(z).P_i\end{aligned}$$

Sums are not essential constructs with the exception of unary sums. τ transitions could be added to the model described above without much difficulty.

To translate a summation from π-Calculus into asynchronous π-Calculus we could use the $(|.|)$ translation.
$(|.|)$ is a homomorphic translation except for sums

$$\begin{aligned}(|\sum_i x_i(z).P_i|) \triangleq \nu\, l \quad & (Proceed\,\langle l\rangle)\\ & \mid \prod_i x_i(z).\nu\, p_i,\ f_i(\bar{l}\,\langle p, f\rangle)\\ & \mid p.(Fail\,\langle l\rangle) \mid (|P_i|))\\ & \mid f.(Fail\,\langle l\rangle)|(\overline{x}_i z)))\end{aligned}$$

Where:

$$\begin{aligned}Proceed(l) &\equiv l(p,f).\overline{p}\\ Fail(l) &\equiv l(p,f).\overline{f}\end{aligned}$$

Note that in the previous equation p_i and f_i are private channels to each process. $\prod$ is used to denote the parallel composition of processes.

Whenever a *send* executes, they will get a $lock$, execute the residual process attached to the *receive* and then pass the lock to the next process. We spawn parallel servers that will determine if a specific process is allowed to proceed or not. Since the *send* is destined to a specific process, one of the parallel process will interact with *Proceed* while the other processes will interact with *Fail*. The processes will keep sending *Fail* to each other, creating a cascading domino effect. At the end, this process will leave a residual *Fail* that can not communicate to any other process.

Ideally we would like to have translations of processes that contains summations of both *send* and *receives* like the one described below:

$$\begin{aligned} P &::= \overline{x}y \mid M \mid P_1|P_2 \mid \nu xP \mid !P \mid N \\ M &::= \sum_i x_i(z).P_i \\ N &::= \sum_i \overline{x_i}d_i.P \end{aligned}$$

When a non-deterministic receive fails, it can send back out the value it received.

$\{\{.\}\}$ is an homomorphic translation except for sums of sends and receives.

First, we translate a sum of receives

$$\sum_i \overline{x_i}d_i.P_i \triangleq \nu s(Proceed\,\langle s\rangle \mid \prod_i \nu a.\overline{x_i}\,\langle d_i, s, a\rangle\,.\nu p,\ f(\overline{a}\,\langle p, f\rangle\,|p.P_i|f.0)$$

$$\begin{aligned} &\textstyle\sum_i y_i(z).qi \triangleq \\ &\qquad \nu r(Proceed\,\langle r\rangle \mid \\ &\qquad\quad \textstyle\prod_i \nu g\ (\overline{g}| \\ &\qquad\qquad !(g.y_i(z,s,a).\nu p_1, f_1(\overline{r}\,\langle p_1,\ f_1\rangle \\ &\qquad\qquad\qquad |P_i(\nu p_2, f_2(\overline{s}\,\langle p_2, f_2\rangle \\ &\qquad\qquad\qquad\quad |p_2.(Fail\,\langle r\rangle)|Fail\,\langle s\rangle\ |Proceed\,\langle a\rangle\,|\{\{Q_i\}\}) \\ &\qquad\qquad\qquad\quad |f_2.((Proceed\,\langle r\rangle\,|Fail\,\langle s\rangle\ |Fail\,\langle a\rangle\,|\overline{g} \\ &\qquad\qquad\qquad\quad |f_1.(Fail\,\langle r\rangle\ \mid \overline{y_1}\,\langle z,s,a\rangle))))) \end{aligned}$$

At the beginning of execution all processes have the lock, and the processes receive channel names on y, z as well as 2 locks a (for this sender) and s for the rest of the receivers. The channels p_1 and f_1 are private channels that allow us the process to check if they should *proceed* or *fail*.

If we get *proceed*, then we have been selected, otherwise the whole communication has failed. In that case we should roll back by repetitevely sending *proceed* and *fails*. It is important to mention that only one copy interacts at any specific time.

The tranlation of a sum of *sends* is left as an exercise to the reader.

5 Conclusion and Final remarks

The final translation was omitted in class, but the reader can refer to [1] for more information.

During this section, we described the importance of Asynchronous π-Calculus, since it properly models real-life situations. We removed extra synchronization by disallowing processes guarded by *send* actions as well as limiting the summations.

During the last part of the section we defined some translations that allows us to move from full π-Calculus into a more leaner version of Asynchronous π-Calculus.

It is still an open-research question to make this translation fully abstracted with respect to the equivalence relations that we have discussed for full π-Calculus (eg. bisimulation). There are ongoing efforts to find the solution to this translations. One of such methods was done by Palamidessi [2] that proposes the following:

Theorem 1. *There does not exist a translation from Asynchronous π-Calculus to π-Calculus with the following properties:*

$$[\![P\sigma]\!] = [\![P]\!]\sigma \tag{1}$$

$$[\![P|Q]\!] = [\![P]\!]|[\![Q]\!] \tag{2}$$

$$\begin{aligned} \forall P,\ N \qquad &\subseteq Fv(P), n \in N, n \notin E, n \notin S \\ &for\ all\ maximal\ computations \\ &if P \xrightarrow{s}{}^{*} \xrightarrow{n} \xrightarrow{t}{}^{*} P', n \in N, n \notin E, n \notin S \\ &[\![P]\!] \xrightarrow{s}{}^{*} \xrightarrow{n} \xrightarrow{t}{}^{*} [\![P']\!] \end{aligned} \tag{3}$$

The first equation states that one might need variable renaming while performing the translation. The second equation establishes that the parallel composition of two processes might result in something different than the parallel composition of the translation of each individual process. The last equation explains that if a process steps from P to P', we might required extra communication actions in order to achieve a translation that makes the equivalent process $[\![P]\!]$step to $[\![P']\!]$.

References

[1] The π-Calculus: A theory of Mobile Processes. Sangiorgi D., Walker D. Cambridge University Press, 2003, 978-0521543279.

[2] Comparing the expressive power of the synchronous and the asynchronous π-Calculus, Palamidessi C, Proc of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 1997

[3] An Object Calculus for Asynchronous Communication. Honda Kohei, Tokoro M. Proc of the European Conference on Object-Oriented Programming (ECOOP), 1991.