Encodings into Asynchronous Pi Calculus

Uwe Nestmann
(Technische Universität Berlin)
Mobility in Pi-Calculus

\[(\nu z)(P \mid R) \mid Q\]

\[P = \overline{x}(z).P'\]
\[Q = x(y).Q'\]

\[P' \mid (\nu z)(R \mid Q')\]

\[x \text{ not in } P'\]
Pi-Calculus

“I reject the idea that there can be a unique conceptual model, or one preferred formalism, for all aspects of something as large as concurrent computation.”
(Robin Milner, 1993)

“Pi Calculus is better than Process Algebra”
(Bill Gates, 2003?)
A Jungle?

many members of the family
Paola Quaglia’s note
“Which Pi Calculus are you talking about?”
π, ν, γ, Λπ, Lπ, Pπ, πI, HΟπ, λπ, πξ, χ, ρ, sπ, κ,
Blue, Fusion, Applied, ...
polyadic, polymorphic, polynomic, polarized, dyadic, ...

many dimensions
communication models (applicability, minimality, ... )
comparisons / encodings
implementations
semantics / models / types / proof techniques / tools
applications
Overview

Part 0a: Encodings vs Full Abstraction
Comparison of Languages/Calculi
Correctness of Encodings

Part 0b: Asynchronous Pi Calculus

Part 1: Input-Guarded Choice
Encoding (Distributed Implementation)
Decoding (Correctness Proof)

Part 2: Output-Guarded Choice
Encoding Separate Choice
Encoding Mixed Choice

Conclusions
Overview

Part 0a: Encodings vs Full Abstraction
Comparison of Languages/Calculi
Correctness of Encodings

Part 0b: Asynchronous Pi Calculus

Part 1: Input-Guarded Choice
Encoding (Distributed Implementation)
Decoding (Correctness Proof)

Part 2: Output-Guarded Choice
Encoding Separate Choice
Encoding Mixed Choice

Conclusions
Comparison of Languages/Calculi
Absolute Expressiveness

(* see also Joachim Parrow @ LIX Colloquium 2006 *)

Given a single process calculus, what can it express?

What objects are expressible
(as closed terms)?

What operators are expressible
(as contexts, or open terms)?

What problems can be solved?
(leader election, matching systems, …)
Relative Expressiveness (I)

Given two (process) calculi $S$ and $T$,

say that “$T$ is as least as expressive as $S$”

to mean that

“$T$ can express anything that $S$” can.
Relative Expressiveness (II)

This can also be formulated without actually saying what is being expressed by exhibiting a (syntactic) encoding \([-]\) : \(S \rightarrow T\)

Such an encoding shall be good/reasonable/compositional/…

Daniele Gorla @ EXPRESS 2006: “Everybody seems to have his/her own idea about which properties to check for.”

both for positive statements (correctness, “goodness”, …) and negative statements (separation, “badness”, …)
\[ \pi\text{-calculus with mixed choice} \]

\[ \pi\text{-calculus with separate choice} \]

- Identity encoding
- \(\pi\text{-calculus with input choice} \) (no output prefix)
  - Nestmann–Pierce 96

- \(\pi\text{-calculus without choice} \) (output prefix)
  - Nestmann 97, Honda–Tokoro 91, Boudol 92

Asynchronous \(\pi\text{-calculus} \)
(no choice, no output prefix)
All of the current proposals of *goodness* are ad-hoc; we do not yet have a proper theory of encodings.
Encodings
Encodings

An encoding is a (total) function

\[
[[\cdot]] : S \to T
\]

that translates

the syntax of language \(S\) (the source) into
the syntax of language \(T\) (the target).

Many encodings are injective, i.e,

\[P \neq Q \implies [[P]] \neq [[Q]]\]

and we only consider compositional definitions,
following the syntactic structure of source terms.
Correctness of Encodings
Indistinguishability (I)

Let P and \([\llbracket P \rrbracket]\) live in the same calculus. The encoding shall be “unnoticable”:

\[
P \cong \llbracket P \rrbracket
\]

• The choice of \(\cong\) captures the expressible artifacts that one considers worth comparing …

• \([[-]]\) and \(\cong\) are closely related.

• Different \textit{results} are often \textit{comparable}.
  \(\Rightarrow\) Seek the strongest equivalence that holds.

• Encodings are often \textit{not injective}.
Indistinguishability (II)

Let $P$ and $[[P]]$ live in two completely different calculi. Then

$$P \cong [[P]]$$

is no longer possible as a requirement.

*Full abstraction* helps !?
Full Abstraction (I)

Notion to capture the quality of denotational models (of programming languages) [Plotkin, TCS 1977].

Let $\mathcal{P}$ be the syntax of a programming language.
Let $\mathcal{D}$ be some mathematical domain.
Let $[[\cdot]] : \mathcal{P} \rightarrow \mathcal{D}$ be the denotational semantics of $\mathcal{P}$.
Let $\approx_\mathcal{P}$ be an (operational) equivalence on $\mathcal{P}$.

Then, $[[\cdot]]$ is called *fully abstract* w.r.t. $\approx_\mathcal{P}$, if

for all $P,Q$ in $\mathcal{P}$: $P \approx_\mathcal{P} Q$ iff $[[P]] = [[Q]]$
Full Abstraction (II)

Let \([[-]] : S \rightarrow T\).

Let \(\equiv_S\) and \(\equiv_T\) be respective equivalences on \(S\) and \(T\).

Then, take \((T, \equiv_T)\) as the respective denotational model, and take the encoding \([[\cdot]]\) as the denotation function.

Then \([[\cdot]]\) is called \textit{fully abstract} w.r.t. \(\equiv_S\) and \(\equiv_T\), if it \textit{preserves and reflects} the equivalences of \(S\) and \(T\):

For all \(P, Q\) in \(S\):

\[ P \equiv_S Q \quad \text{iff} \quad [[P]] \equiv_T [[Q]] \]
Full Abstraction (III)

Problems:
• on what basis to choose $\cong_S$ and $\cong_T$ (cf. Rob van Glabbeek’s talk)
• various ways to have results for congruences
  – all target contexts
  – only translated contexts (respecting the protocol)
  – only well-typed contexts (w.r.t. a target type system)

Observation:
• full abstraction results are “easy to get”
• full abstraction results are hard to compare
**Operational Correspondence (I)**

**Completeness (Preservation of execution step).**

If $S \xrightarrow{s} S'$, then $[S] \Rightarrow_t [S']$

**Soundness (Reflection of execution steps).**

If $[S] \Rightarrow_t [S']$ then $S \Rightarrow_s S'$

If $[S] \xrightarrow{t} T$ then there is $S \xrightarrow{s} S'$ such that $T \subseteq_t [S']$

If $[S] \Rightarrow_t T$ then there is $S \Rightarrow_s S'$ such that $T \Rightarrow_t [S']$
Operational Correspondence (II)

Obviously (?), some form of operational correspondence is often employed as a means to support the proof of full abstraction w.r.t. bisimulation-based equivalences.

Think about how to prove:

\[ P \approx_{S} Q \quad \text{iff} \quad [[P]] \approx_{T} [[Q]] \]
Question

Assume an encoding $[[ - ]] : S \rightarrow T$.
Assume $\equiv_{S1}$ and $\equiv_{S2}$ are equivalences on $S$.
Assume $\equiv_{T1}$ and $\equiv_{T2}$ are equivalences on $T$.
Assume $\equiv_{S1} \subseteq \equiv_{S2}$ and $\equiv_{T1} \subseteq \equiv_{T2}$.

Does $\text{F.A.w.r.t.} (\equiv_{S1}, \equiv_{T1})$ imply $\text{F.A.w.r.t.} (\equiv_{S2}, \equiv_{T2})$?

- **Special case**: consider universal relations
- **Special case**: identity embeddings
Assume an arbitrary encoding $[[ - ]] : S \rightarrow T$.

Then, $[[ - ]]$ is fully abstract w.r.t. $(\text{Ker}( [[ - ]] ), \text{Id}_T)$. 
Task

Assume an encoding \([\cdot - \cdot]\) : \(S \rightarrow T\).
Assume \(\equiv_{S1}\) and \(\equiv_{S2}\) are equivalences on \(S\).
Assume \(\equiv_{T1}\) and \(\equiv_{T2}\) are equivalences on \(T\).

Identify “reasonable” conditions to state that:

\[
\text{F.A.w.r.t. (} \equiv_{S1}, \equiv_{T1} \text{) is better than F.A.w.r.t. (} \equiv_{S2}, \equiv_{T2} \text{)}
\]
Example Encodings
Tuples

\[
\overline{x}\langle a_1 \ldots a_n \rangle. Q] = \nu w \overline{x} w. \overline{w} a_1. \ldots. \overline{w} a_n. [Q]
\]

\[
\overline{x}(z_1 \ldots z_n). P] = x(w). w(z_1). \ldots. w(z_n). [P]
\]

... all other operators translated homomorphically

- operational correspondence
- no “obvious” F.A.
- F.A. w.r.t. translated contexts
- F.A. w.r.t. typed barbed congruence
Functions as Processes

\[
\begin{align*}
\left[ \lambda x. M \right]_p & \overset{\text{def}}{=} \nu y \overline{p}\langle y \rangle . !y(x,q). \left[ M \right]_q \\
\left[ x \right]_p & \overset{\text{def}}{=} \overline{x}\langle p \rangle \\
\left[ M N \right]_p & \overset{\text{def}}{=} \nu q \left( \left[ M \right]_q \mid q(v) . \nu x \left( \overline{v}\langle x,p \rangle . !x(r). \left[ N \right]_r \right) \right)
\end{align*}
\]

various 1/2 F.A. results for various variants
used to compare Lambda and Pi Calculi
inspired Lambda Calculus theory
used to prove new equations
Objects as Processes

<table>
<thead>
<tr>
<th>Expression</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>([ a.\text{clone}]^k_p)</td>
<td>(\nu q \left( \left[ a \right]_q^k \mid q(y, k') \cdot \overline{y}\langle \text{cln}_p, k' \rangle \right))</td>
</tr>
<tr>
<td>([ a.\text{alias}(b)]^k_p)</td>
<td>(\nu q x y \left( \left[ a \right]<em>q^k \mid q_y(y, k_y) \cdot \left[ b \right]</em>{q_x}^k \mid q_x(x, k_x) \cdot \overline{y}\langle \text{ali}_p \langle x, p \rangle, k_x \rangle \right) )</td>
</tr>
<tr>
<td>([ a.l_j \triangleleft \varsigma(s, \tilde{x})b]^k_p)</td>
<td>(\nu q \left( \left[ a \right]_q^k \mid q(y, k').(\nu t) \left( ! t(s, \tilde{x}, r, k). \left[ b \right]_r^k \mid \overline{y}\langle \text{upd}_j \langle t, p \rangle, k' \rangle \right) \right) )</td>
</tr>
<tr>
<td>([ a.l_j \langle a_1 \ldots a_n \rangle]^k_p)</td>
<td>(\nu q q_1 \ldots q_n \left( \left[ a \right]<em>q^k \mid q(y, k_0) \cdot \left[ a_1 \right]</em>{q_1}^{k_0} \mid q_1(x_1, k_1) \cdot \left[ a_2 \right]<em>{q_2}^{k_1} \mid \ldots q_n(x_n, k_n) \cdot \overline{y}\langle \text{inv}</em>{j-1}(x_1 \ldots x_n, p), k_n \rangle \cdots \right) )</td>
</tr>
<tr>
<td>([ a.\text{surrogate}]^k_p)</td>
<td>(\nu q \left( \left[ a \right]_q^k \mid q(y, k') \cdot \overline{y}\langle \text{sur}_p, k' \rangle \right) )</td>
</tr>
<tr>
<td>([ a.\text{ping}]^k_p)</td>
<td>(\nu q \left( \left[ a \right]_q^k \mid q(y, k') \cdot \overline{y}\langle \text{ping}_p, k' \rangle \right) )</td>
</tr>
<tr>
<td>([ \text{let } x = a \text{ in } b]^k_p)</td>
<td>(\nu q \left( \left[ a \right]_q^k \mid q(x, k') \cdot \left[ b \right]_p^{k'} \right) )</td>
</tr>
<tr>
<td>([ x]^k_p)</td>
<td>(\overline{p}\langle x, k \rangle )</td>
</tr>
<tr>
<td>([ \text{fork}(a)]^k_p)</td>
<td>(\nu q t \left( \left[ a \right]_q^{k'} \mid \overline{p}\langle t, k \rangle \mid q(x, k').t(r, k'').\overline{r}\langle x, k'' \rangle \right) )</td>
</tr>
<tr>
<td>([ \text{join}(b)]^k_p)</td>
<td>(\nu q \left( \left[ b \right]_q^k \mid q(t, k') \cdot \overline{t}\langle p, k' \rangle \right) )</td>
</tr>
</tbody>
</table>

Table 6: Translational semantics of Øjeblik — Clients, Scoping, Concurrency
used to provide a formal semantics used to prove an equation correct used for debugging an existing compiler
Overview

Part 0a: Encodings vs Full Abstraction
Comparison of Languages/Calculi
Correctness of Encodings

Part 0b: Asynchronous Pi Calculus

Part 1: Input-Guarded Choice
Encoding (Distributed Implementation)
Decoding (Correctness Proof)

Part 2: Output-Guarded Choice
Encoding Separate Choice
Encoding Mixed Choice

Conclusions
Send operations are **non-blocking**.

Thus, there is neither need nor use to consider send prefixes.

Instead, include just *messages ... without continuation*. 
Syntax

Let \( \mathbb{N} \) be a countable set of names. Then, the set \( \mathbb{P} \) of processes \( P \) is defined by

\[
P ::= (x) \; P \mid P \mid P \mid yz \mid 0 \mid R \mid !R \\
R ::= y(x) \cdot P
\]
Operational Semantics (I)

\[ \mu ::= \bar{y}(z) \mid \bar{y}z \mid yz \mid \tau \]

\( OUT:\) \quad \bar{y}z \xrightarrow{\bar{y}z} 0

\( INP:\) \quad y(x).P \xrightarrow{yz} P\{z/x\}

\( R-INP:\) \quad !y(x).P \xrightarrow{yz} P\{z/x\} \mid !y(x).P

\( COM^*_1:\) \quad \frac{P_1 \xrightarrow{\bar{y}z} P'_1 \quad P_2 \xrightarrow{yz} P'_2}{P_1 \parallel P_2 \xrightarrow{\tau} P'_1 \parallel P'_2}
**Operational Semantics (II)**

### OPEN:

\[
P \xrightarrow{\bar{y}z} P' \\
(z) \quad P \xrightarrow{\bar{y}(z)} P'
\]

if \( y \neq z \)

### CLOSE\(_1^*\):

\[
P_1 \xrightarrow{\bar{y}(z)} P'_1 \quad P_2 \xrightarrow{yz} P'_2
\]

if \( z \notin \text{fn}(P_2) \)

\[
P_1 \parallel P_2 \xrightarrow{\tau} (z)(P'_1 \parallel P'_2)
\]
TABLE 1

<table>
<thead>
<tr>
<th>OPERATIONAL SEMANTICS (III)</th>
</tr>
</thead>
</table>

**PAR\textsuperscript{*}:**

\[
P_1 \xrightarrow{\mu} P'_1 \quad \frac{P_1 | P_2 \xrightarrow{\mu} P'_1 | P_2}{\text{if } \text{bn}(\mu) \cap \text{fn}(P_2) = \mathcal{0}}
\]

**RES:**

\[
\frac{P \xrightarrow{\mu} P'}{(x)P \xrightarrow{\mu} (x)P'} \quad \text{if } x \notin \text{n}(\mu)
\]

**ALPHA:**

\[
\frac{\hat{P} \xrightarrow{\mu} \hat{P}'}{P \xrightarrow{\mu} \hat{P}'} \quad \text{if } P = \alpha \hat{P}
\]
**Asynchronous Bisimulation**

**Definition**

A binary relation $\mathcal{S}$ on processes is a *strong simulation* if $(P, Q) \in \mathcal{S}$ implies:

- if $P \xrightarrow{\mu} P'$, where $\mu$ is either $\tau$ or output with $\text{bn}(\mu) \cap \text{fn}(P \mid Q) = \emptyset$, there is $Q'$ such that $Q \xrightarrow{\mu} Q'$ and $(P', Q') \in \mathcal{S}$

- $(\tilde{a}z \mid P, \tilde{a}z \mid Q) \in \mathcal{S}$ for arbitrary messages $\tilde{a}z$.

$B$ is called a *strong bisimulation* if both $B$ and $B^{-1}$ are strong simulations.

Replacing $Q \xrightarrow{\mu} Q'$ with $Q \xrightarrow{\hat{\mu}} Q'$ ...

... yields the *weak* versions.
Definition

A weak simulation $\mathcal{S}$ is called

- **progressing**, if $P \xrightarrow{\mu} P'$ implies that there is $Q'$ with $Q \xrightarrow{\mu} Q'$ **at least one** $\tau$

- **strict**, if $P \xrightarrow{\mu} P'$ implies that there is $Q'$ with $Q \xrightarrow{\hat{\mu}} Q'$ **at most one** $\tau$

for all $(P, Q) \in \mathcal{S}$ and for all $\mu$ being $\tau$ or output with $\text{bn}(\mu) \cap \text{fn}(P \mid Q) = \emptyset$. 

(*) better control on $\tau$-moves *)
Too Weak (I): Expansion

[Arun-Kumar, Hennessy 1991]

**Definition**

A binary relation $\mathcal{E}$ on processes is an expansion if $\mathcal{E}$ is a progressing simulation and $\mathcal{E}^{-1}$ is a strict simulation.

$\sim \subset \preceq \subset \simeq$
Too Weak (II): Eventual Progress

**Definition**

A weak simulation $\mathcal{S}$ is *eventually progressing* if, for all $(P, Q) \in \mathcal{S}$, there is a natural number $k_P \in \mathbb{N}$ such that $P \xrightarrow{\tau}^n P'$ with $n > k_P$ implies that there is $Q'$ with $Q \xrightarrow{\tau} + Q'$ such that $(P', Q') \in \mathcal{S}$.

*Eventually progressing simulations preserve divergence.*
Structural Laws

\[\text{\textbf{\textit{\textit{TABLE 2}}}}\]

Structural Laws:

- \(\alpha\)-conversion: \(P \equiv Q \) if \( P = \alpha Q \)
- Associativity: \( P \mid (Q \mid R) \equiv (P \mid Q) \mid R \)
- Commutativity: \( P \mid Q \equiv Q \mid P \)
- Neutrality: \( P \mid 0 \equiv P \)
- Scope Extrusion: \((y) P \mid Q \equiv (y)(P \mid Q)\) if \( y \notin \text{fn}(Q)\)
- Scope Elimination: \((y) Q \equiv Q\) if \( y \notin \text{fn}(Q)\)
When Weak is Too Strong ...

\[ P = (i) \text{ ( } \bar{i} \mid i.A \mid i.B \mid i.C) \]

\[ Q = (i_1)(i_2) \text{ ( } \bar{i}_1 \mid \bar{i}_2 \mid i_1.A \mid i_1.(i_2.B \mid i_2.C) ) \]

\( P \) and \( Q \) are not weakly bisimilar. They just simulate each other. However, we can do much better ...

\[ \]
Coupled Simulation (I)

[Parrow, Sjödin 1994]

**Definition**

A *mutual simulation* is a pair \((\mathcal{S}_1, \mathcal{S}_2)\), where \(\mathcal{S}_1\) and \(\mathcal{S}_2^{-1}\) are weak simulations.

A *coupled simulation* is a mutual simulation \((\mathcal{S}_1, \mathcal{S}_2)\) satisfying

- if \((P, Q) \in \mathcal{S}_1\), then there is some \(Q'\) such that \(Q \Rightarrow Q'\) and \((P, Q') \in \mathcal{S}_2\)
- if \((P, Q') \in \mathcal{S}_2\), then there is some \(P'\) such that \(P \Rightarrow P'\) and \((P', Q') \in \mathcal{S}_1\)

\(P \leftrightarrow Q\) \hspace{1cm} *coupled simulation equivalent* (or coupled similar)
Coupled Simulation (II)

- if \((P, Q) \in S_1\), then there is some \(Q'\) such that \(Q \Rightarrow Q'\) and \((P, Q') \in S_2\)
- if \((P, Q') \in S_2\), then there is some \(P'\) such that \(P \Rightarrow P'\) and \((P', Q') \in S_1\)
Coupled Simulation (III)

\[
\begin{array}{c}
\begin{array}{ccc}
P & \equiv & P' \\
\equiv & \equiv & \equiv \\
\approx & \subset & \iff \\
\approx & \subset & \iff ,
\end{array}
\end{array}
\]

\[\iff \text{ is a congruence on } P.\]
Overview

Part 0a: Encodings vs Full Abstraction
  Comparison of Languages/Calculi
  Correctness of Encodings

Part 0b: Asynchronous Pi Calculus

Part 1: Input-Guarded Choice
  Encoding (Distributed Implementation)
  Decoding (Correctness Proof)

Part 2: Output-Guarded Choice
  Encoding Separate Choice
  Encoding Mixed Choice

Conclusions
Choice in Asynchronous Contexts

The intuition of asynchronous send operations is that the \textit{send actions happen instantaneously}, without interference with the environment.

Having such a send compete with an receive, which itself is to wait for the availability of a message, will always make the send action win over the receive action.

So, \textit{better not mix up} send and receive within a choice ...
The set $\mathbb{P}^\Sigma$ of processes with *input-guarded choice*

$$P ::= (x) \ P \mid P \mid P \mid \bar{y}z \mid 0 \mid R \mid !R \mid \sum_{j \in J} R_j$$

$$R ::= y(x) \cdot P$$

\[ \sum_{j \in J} y_j(x) \cdot P_j \xrightarrow{y_kz} P_k \quad \text{if} \quad k \in J \]
Encoding Choice
Two Choice Encodings (I)

\[
S ::= P^* \quad \vdash \quad P \quad \vdash \quad P.
\]

For notational convenience, the target language contains:

\[
P ::= \cdots | \text{test } y \text{ then } P \text{ else } P.
\]

\[
\begin{align*}
\text{with:} & \quad \text{test } l \text{ then } P_1 \text{ else } P_2 & \xrightarrow{lt} & P_1 \\
& \quad \text{test } l \text{ then } P_1 \text{ else } P_2 & \xrightarrow{lf} & P_2
\end{align*}
\]

(* for the special names t and f *)
Two Choice Encodings (II)

\( C[\ ] : S \rightarrow T \)

\[
\begin{align*}
\left[ (x) \ P \right] & \overset{\text{def}}{=} (x)\left[ P \right] \\
\left[ y z \right] & \overset{\text{def}}{=} y z \\
\left[ y(x) \right] & \overset{\text{def}}{=} y(x) \cdot \left[ P \right] \\
\left[ P_1 | P_2 \right] & \overset{\text{def}}{=} \left[ P_1 \right] | \left[ P_2 \right] \\
\left[ 0 \right] & \overset{\text{def}}{=} 0 \\
\left[ ! R \right] & \overset{\text{def}}{=} !\left[ R \right].
\end{align*}
\]

\[
\left[ \sum_{j \in J} R_j \right] \overset{\text{def}}{=} (l) \left( \bar{t} \mid \prod_{j \in J} \text{Branch}_l \langle \left[ R_j \right] \rangle \right)
\]
Two Choice Encodings (III)

\[
C \left[ \sum_{j \in J} R_j \right] \overset{\text{def}}{=} (l) \left( \bar{t}t \mid \prod_{j \in J} \text{Read}_l \langle C \left[ R_j \right] \rangle \right)
\]

\[
\text{Read}_l \langle R \rangle \overset{\text{def}}{=} y(x) \cdot \text{Test}_l \langle R \rangle
\]

\[
\text{Test}_l \langle R \rangle \overset{\text{def}}{=} \text{test } l \text{ then } \text{Commit}_l \langle R \rangle \text{ else } \text{Abort}_l \langle R \rangle
\]

\[
\text{Commit}_l \langle R \rangle \overset{\text{def}}{=} \bar{f}f \mid P
\]

\[
\text{Abort}_l \langle R \rangle \overset{\text{def}}{=} \bar{f}f \mid \bar{y}x
\]
Two Choice Encodings (IV)

\[ \mathcal{D} \left[ \sum_{j \in J} R_j \right] \overset{\text{def}}{=} (l) \left( \bar{l}t \mid \prod_{j \in J} \text{Read}_i \langle \mathcal{D} [R_j] \rangle \right) \]

\[ \text{Read}_i \langle R \rangle \overset{\text{def}}{=} y(x).\text{Test}_i \langle R \rangle \]

\[ \text{Test}_i \langle R \rangle \overset{\text{def}}{=} \text{test } l \text{ then Commit}_i \langle R \rangle \text{ else Abort}_i \langle R \rangle \]

\[ \text{Commit}_i \langle R \rangle \overset{\text{def}}{=} \bar{l}f \mid P \]

\[ \text{Abort}_i \langle R \rangle \overset{\text{def}}{=} \bar{l}f \mid \bar{y}x \]

\[ \text{Test}_i \langle R \rangle \overset{\text{def}}{=} \text{test } l \text{ then Commit}_i \langle R \rangle \oplus \text{Undo}_i \langle R \rangle \text{ else Abort}_i \langle R \rangle \]

\[ \text{Undo}_i \langle R \rangle \overset{\text{def}}{=} \bar{l}t \mid \bar{y}x. \]
Encoding an Example Process (I)

\[ S = \overline{y}_2 z \mid N \quad \text{where} \quad N = R_1 + R_2 \quad \text{with} \quad R_i = y_i(x).P_i \]

\[ S : \quad \begin{array}{c}
N \xleftarrow{\overline{y}_2 z} S \\
S \xrightarrow{\tau} P_2\{z/x\}
\end{array} \]

\[ S \not\cong C[S] \]

\[ \equiv \underbrace{(l)(\overline{f} \mid B_1)}_{\approx 0} \mid C[P_2]\{z/x\} \]
Encoding an Example Process (II)

\[ S = \overline{y_2}z \mid N \quad \text{where} \quad N = R_1 + R_2 \quad \text{with} \quad R_i = y_i(x).P_i \]

\[ \mathcal{D}[N] \xleftarrow{\overline{y_2}z} \mathcal{D}[S] \xrightarrow{\tau} D \xrightarrow{\tau} D' \xrightarrow{\tau} D_2 \]

\[ S \approx \mathcal{D}[S]. \]

\[ \equiv (l)(\overline{f} \mid B_1 \mid B_2) \mid \mathcal{D}[P_2] \{z/x\} \approx 0 \]
Expanded Encodings

\( S = \) \( \mathcal{P}_\Sigma \)

\( \mathcal{C}[\{} \), \( \mathcal{D}[\{} \)

\( \mathcal{P} \downarrow \)

\( \mathcal{P}^{\text{test}} \)

\( \mathcal{B}[\{} \)

essentially the “classical” encoding of booleans

\[ \text{for all } S \in \mathcal{S} : \text{ for all } \mathcal{C}[S] \rightarrow ^* T : T \preceq \mathcal{B}[T] \]
Correctness Proof
Correctness Proof Strategy (I)

Provide a notational framework that allows to *decode* any derivative of an encoded term to “the” corresponding state in the source.

\[ S := p_S \]

\[ p_S \rightarrow p_{\text{test}} \]

\[ =: T \]
Correctness Proof Strategy (II)

For the encoding with partial commitments, we have to denote two corresponding states!
Factorization
The Decoding Problem

Once a source term is translated into the target language, its source-level structure is often completely lost.

\[
\left[ \sum_{j \in J} R_j \right] \overset{\text{def}}{=} (l) \left( \bar{t} \mid \prod_{j \in J} \text{Branch}_l \left[ \left[ R_j \right] \right] \right)
\]

\[
\begin{array}{ccc}
S & \overset{U_0[ ]}{\rightarrow} & A \\
U_0[ ] & & \Uparrow \\
\downarrow & & \\
B[ ] & \rightarrow & T
\end{array}
\]
Intermediate Language (I)

**annotated choice**

$$\left( \sum_{j \in J} R_j \right)^v_B$$

- a partial function $$v : J \rightarrow V$$
- a possibly empty set $$B \subseteq J$$
  $$B \cap \text{dom}(v) = \emptyset$$

**corresponding to decoded source-level choice states:**

- **Read-state** if $$k \in J \setminus (V \cup B)$$
- **Test-state** if $$k \in V$$
- **Commit/Abort-state** if $$k \in B.$$
**Definition 5.2.1 (Annotated choice).** Let \( J \) be a set of indices. Let \( \mathbf{R}_j = y_j(x) \). Let \( \mathbf{P}_j \) be input prefixes for \( j \in J \). Let \( \mathbf{v} : J \rightarrow (V \cup \mathbf{B}) \) and \( \mathbf{v} \{ v_k / x \} = \mathbf{v} - k \). Then, \( \mathbf{v} \) are referred to as bare and annotated choice, respectively.

Annotated choice is given the operational semantics in Table 5. The dynamics of annotated choice mimic precisely the behavior of the intended low-level process. \( \text{READ} \) allows a branch \( k \) in \( \text{Read-state} \) (\( k \in J \backslash (V \cup B) \)) to optimistically consume a message. If the choice is not yet resolved (\( B = \mathbf{v} - k \)), \( \text{COMMIT} \) specifies that an arbitrary branch \( k \) in \( \text{Test-state} \) (\( k \in V \)) can immediately evolve into its Commit-state, i.e., trigger its continuation process \( \mathbf{P}_k \). After the choice is resolved (\( B \{ v_k / x \} = \mathbf{v} - k \)), \( \text{ABORT} \) allows any branch \( k \) in \( \text{Commit-state} \) (\( k \in V \)) to evolve into its Abort-state to release their consumed messages. Intuitively, by reading the lock, a branch immediately leaves the choice system and exits. Therefore, annotated choice only contains branches in either \( \text{Read-} \) or \( \text{Test-} \) state.

We distinguish three cases for choice constructors that are important enough to give them names: initial for \( \mathbf{v} = \mathbf{v} - k = \mathbf{B} \), partial for \( \mathbf{v} \{ v_k / x \} = \mathbf{v} - k \) and \( \mathbf{B} = \mathbf{v} - k \), and committed for \( \mathbf{B} \{ v_k / x \} = \mathbf{v} - k \). Note that both initial and partial choice contain all branches, whereas committed choice never does; it will even become empty, once all branches have reached their final state (\( B = J \)).

Committed choice exhibits a particularly interesting property: its branches in \( \text{Test-state} \) already have consumed a message which they will return after recognizing, by internally testing the lock, that the choice is already committed; its branches in \( \text{Read-state} \) are still waiting for values to be consumed and \( \text{NAK/NAK} \) since the choice is resolved and the lock carries \( \text{f/NAK/NAK} \) immediately resent after an internal step. Processes with such receive-and-resend behavior are weakly bisimilar to \( \mathbf{0} \) and were called \( \mathbf{27} \).
Annotated Choice States

(* as composed by the states of its branches *)

initial for $V = \emptyset = B$

partial for $V \neq \emptyset$ and $B = \emptyset$

committed for $B \neq \emptyset$
The Key Lemma

(* the essence of asynchrony *)

\[
\left( \sum_{j \in J} R_j \right)^v_{B \neq \emptyset} \geq \prod_{j \in V} y_j v_j.
\]

Messages consumed (see \(v\)) by committed choices (see \(B\)) are still fully available to the environment.

\[
\left( \sum_{j \in J} R_j \right)_{B \neq \emptyset} \geq 0
\]

“asynchronous garbage”
Intermediate Encoding

$S \xrightarrow{A[\ ]} A \in \mathcal{P}(\Sigma)$

$C[\ ] \xrightarrow{\Delta} \top$

$\mathcal{A} \left[ \sum_{j \in J} R_j \right] \overset{\text{def}}{=} \left( \sum_{j \in J} \mathcal{A}[R_j] \right) \emptyset

$\mathcal{A}$ is a weak simulation up to expansion.
Flattening

\[ \mathcal{F} \left[ \left( \sum_{j \in J} R_j \right)^v \right] \overset{\text{def}}{=} (l) \left( \overline{l}b \mid \prod_{j \in J \setminus (V \cup B)} \text{Read}_1 \langle \mathcal{F} \left[ R_j \right] \rangle \right) \]

\[ \mid \prod_{j \in V} \text{Test}_1 \langle \mathcal{F} \left[ R_j \right] \rangle \{ v(j) / x \} \]

where \( b \) is \( t \), if \( B \neq \emptyset \), and \( f \), otherwise

\( \mathcal{F} \) is a strong bisimulation.
Decoding
Decoding (I)

Decoding (Section 5.4) Annotated source terms deal with partial commitments explicitly. We define two decoding functions $U_\#[ ]$ and $U_\bot[ ]$ of annotated terms back into source term where $U_\#/GS$ resets and $U_\bot$ completes partial commitments.

Expansion (Section 5.5) Since the target language $T$ is a language extended with Booleans, we show that their use in our setting is rather well behaved according to the expanding encoding $B$. The factorization, the decodings, and the expansion enjoy several nice properties:

1. $F$ is a strong bisimulation between abbreviations and Boolean target terms.
2. $(U_\#/GS, U_\bot)$ is a coupled simulation between abbreviations and source terms.
3. $B$, i.e., a variant of it, is an expansion for Booleans in target terms.

Those can be combined to provide a coupled simulation on $S_P$. The observation that every source term $S$ and its translation $C/ETB/S/EM$ are related by this relation concludes the proof of coupled-simulation-correctness of the $C/ETB$-encoding (Section 5.6).

Simplifications due to homomorphic encodings and decodings. Many of the proofs in this section have in common that they exhibit particular transitions of terms by constructing appropriate inference trees either from the inductive structure of (annotated) terms, or by simply replacing some leafs in the inference trees of their encodings or decodings.

Since the $A, F, U_\#/GS, U_\bot,$ and $B$-functions are each defined homomorphically on every constructor of $P$ according to the scheme in Section 4, there is a strong syntactic correspondence between terms and their respective translations, and, as a consequence, there is also a strong correspondence between transition inference trees. More precisely, since in transitions involving choice there is at most one application of a choice rule, and an application of the choice rule always represents a leaf in the inference tree, it suffices for all proofs to regard choice terms in isolation.
Annotated Choice States

(* as composed by the states of its branches *)

initial for $V = \emptyset = B$

partial for $V \neq \emptyset$ and $B = \emptyset$

committed for $B \neq \emptyset$
Decoding (II)

\[
\begin{align*}
\text{initial:} & \quad \mathcal{U} \left[ \left( \sum_{j \in J} R_j \right)^\varnothing \right] \overset{\text{def}}{=} \sum_{j \in J} \mathcal{U} [R_j] \\
\text{committed:} & \quad \mathcal{U} \left[ \left( \sum_{j \in J} R_j \right)^v \right] \overset{\text{def}}{=} \prod_{j \in V} \overline{y}_j v_j \\
\text{partial:} & \quad \mathcal{U}_b \left[ \left( \sum_{j \in J} R_j \right)^v \right] \overset{\text{def}}{=} \prod_{j \in V} \overline{y}_j v_j \mid \sum_{j \in J} \mathcal{U}_b [R_j] \\
\text{partial:} & \quad \mathcal{U}_# \left[ \left( \sum_{j \in J} R_j \right)^v \right] \overset{\text{def}}{=} \overline{y}_j v_j \mid \mathcal{U}_# [P_k] \{v_k/x\} \\
& \quad k := \text{take}(V)
\end{align*}
\]
Results (I)

\[ \mathcal{U}_b \] is a weak simulation.
\[ \mathcal{U}^{-1}_\# \] is a weak simulation.

\[ \mathcal{U}_b \] is strict; \[ \mathcal{U}^{-1}_\# \] is progressing.

Both \[ \mathcal{U}_b \] and \[ \mathcal{U}^{-1}_\# \] are eventually progressing.
Results (II)

(\mathcal{U}_b, \mathcal{U}_\#) \text{ is a coupled simulation.}

(\mathcal{U}_\#^{-1}, \mathcal{U}_b^{-1}) \text{ is a coupled simulation.}

\(\mathcal{U}_b[A] \xrightarrow{\gamma} \mathcal{U}_\#[A] \xrightarrow{\gamma} \mathcal{U}_\#[A] = \mathcal{U}_\#[A'] = \mathcal{U}_b[A']\)
\((\mathcal{C}, \mathcal{G})\) is a coupled simulation.

For all \(S \in \mathcal{G}: (S, \mathcal{C} \leadsto [S]) \in \mathcal{C} \cap \mathcal{G}.

For all \(S \in \mathcal{G}: S \rightharpoonup \mathcal{C} \leadsto [S].\)
Results (IV)

Running the same proof strategy for the atomically committing encoding, we may use a single decoding functions.

\( U \) is a weak bisimulation.

For all \( S \in \mathbb{S} : S \approx D[S] \).


\( \mathcal{S}^{-1} \) is eventually progressing.

\( \mathcal{C} \) is divergence-free.
Operational Correspondence

(* for free *)

\[ S \xrightarrow{a_1 \cdots a_n} S' \xrightarrow{\epsilon} T \xrightarrow{a_1 \cdots a_n} \mathcal{C} \prec [S''] \]

\[ \mathcal{C} \prec [S] \xrightarrow{a_1 \cdots a_n} T \xrightarrow{\epsilon} \mathcal{C} \prec [S''] \]
Intermediate
Conclusions
A bit of cheating: encodings depend on n-ary choice.

The role of name-passing.

Asynchronous Pi is “sufficiently” expressive ... ... for programming. (Who needs mixed choice?)

The role of coupled simulation.

There is no theory of encodings yet.
Overview

Part 0a: Encodings vs Full Abstraction
Comparison of Languages/Calculi
Correctness of Encodings

Part 0b: Asynchronous Pi Calculus

Part 1: Input-Guarded Choice
Encoding (Distributed Implementation)
Decoding (Correctness Proof)

Part 2: Output-Guarded Choice
Encoding Separate Choice
Encoding Mixed Choice

Conclusions
Calculus
Choice in Synchronous(!) Contexts

As motivated by Catuscia Palamidessi's talk, we focus now on encodings of choice operators that may contain (synchronous) send prefixes.

We try to investigate some limits of encodability. In contrast to Catuscia Palamidessi, we seek positive correctness results.
The \( \pi \)-calculus hierarchy.

The dashed line represents an identity encoding.

- \( \pi \)-calculus with mixed choice
  - Palamidessi 97

- \( \pi \)-calculus with separate choice
  - Identity encoding
  - \( \pi \)-calculus with input choice (no output prefix)
    - Nestmann–Pierce 96
  - Nestmann 97

- \( \pi \)-calculus without choice (output prefix)
  - Honda–Tokoro 91, Boudol 92

- Asynchronous \( \pi \)-calculus (no choice, no output prefix)
  - Identity encoding

Wednesday, October 14, 2009
Syntax + Semantik

\[ \pi : ::= y!\tilde{z} \quad \mid \quad y?\tilde{x} \]

\[ \text{\$\text{inp}: P ::=... \quad \mid \quad \sum_{i \in I} y_i?\tilde{x}_i.P_i} \]

\[ \text{T: P ::=... \quad \mid \quad y?\tilde{x}.P} \quad \mid \quad y!\tilde{z}. \]
Reduction Semantics

\[ S^\text{mix}, S^\text{sep} : \ (\cdots + y?[^\tilde{x}].P) \mid (y![^\tilde{z}].Q + \cdots) \rightarrow P\{^\tilde{z} / ^\tilde{x}\} \ | \ Q \]

\[ S^\text{mix}, S^\text{sep} : \ y^*[^\tilde{x}].P \mid (y![^\tilde{z}].Q + \cdots) \rightarrow P\{^\tilde{z} / ^\tilde{x}\} \ | \ Q \mid y^*[^\tilde{x}].P \]

\[ S^\text{inp} : \ (\cdots + y?[^\tilde{x}].P) \mid y![^\tilde{z}] \rightarrow P\{^\tilde{z} / ^\tilde{x}\} \]

\[ S^\text{inp}, T : \ y^*[^\tilde{x}].P \mid y![^\tilde{z}] \rightarrow P\{^\tilde{z} / ^\tilde{x}\} \mid y^*[^\tilde{x}].P \]

\[ T : \ y?[^\tilde{x}].P \mid y![^\tilde{z}] \rightarrow P\{^\tilde{z} / ^\tilde{x}\} \]

\[ T : \ \text{test } y \text{ then } P \text{ else } Q \mid y!^t \rightarrow P \]

\[ T : \ \text{test } y \text{ then } P \text{ else } Q \mid y!^f \rightarrow Q \]

\[ \text{if } \ P \rightarrow P' \ \text{ then } \ (\nu x) \ P \rightarrow (\nu x) \ P' \]

\[ \text{if } \ P \rightarrow P' \ \text{ then } \ Q \mid P \rightarrow Q \mid P' \]

\[ \text{if } \ P \equiv Q \rightarrow Q' \equiv P' \text{ then } \]

\[ P \mid Q \equiv Q \mid P \]

\[ (\nu y) (\nu x) \ P \equiv (\nu x) (\nu y) \ P \]

\[ P \mid (Q|R) \equiv (P|Q) \mid R \]

\[ (\nu y) P \mid Q \equiv (\nu y) (P \mid Q) \text{ if } y \notin \text{fn}(Q) \]
Encodings
Homomorphic Encoding Scheme

\[
\begin{align*}
[P_1 | P_2] & \overset{\text{def}}{=} [P_1] | [P_2] \\
(\nu x) P & \overset{\text{def}}{=} (\nu x) [P] \\
\sum_{i \in I} \pi_i.P_i & \overset{\text{def}}{=} (\nu l)(l![t] | \prod_{i \in I} [\pi_i.P_i]_l)
\end{align*}
\]

Encoding Input-Guarded Choice:

\[
\begin{align*}
[y![\tilde{z}]] & \overset{\text{def}}{=} y![\tilde{z}] \\
[y?[^{\tilde{x}}]P]_l & \overset{\text{def}}{=} y?[^{\tilde{x}}] \cdot \text{test } l \text{ then } (l![f] | [P]) \text{ else } (l![f] | y?[^{\tilde{x}}]) \\
[y*[^{\tilde{x}}]P] & \overset{\text{def}}{=} y*[^{\tilde{x}}].[P]
\end{align*}
\]

The locking game on \( l \) does the trick.
FIG. 5. S
sep/DC4T.

either enabling the sender's continuation to proceed, or to abort it. Since output-
guards are also branches in a choice whose state must be tested, the corresponding
lock is, in addition to $z$ and $a$, transmitted to some matching receiver that then
performs the required choice-test.

Input-guards, revisited. The encoding is more elaborate due to the increased
information that is transmitted by send-requests. First, there are now two locks that
have to be tested in some order. In Fig. 5, we chose to test the
local lock $l$ first and, only in the case of a positive outcome, to test the
remote lock $r$. (This particular order is useful in an actual distributed implementation, where remote communica-

tion is usually much more expensive than local communication.) Second, we have
to use the acknowledgment channel correctly, which means that a positive acknowl-
edgment may only be sent if both locks were tested positively. Third, in the case
that the test of the sender's choice-lock was negative, we must not resend the send-
request $/\text{NAK}$/NAK instead, and only if the test of the receiver's choice-lock was positive, we
have to restart the receiver process from the beginning by allowing it to try other
send-requests. In Fig. 5, this is implemented by recursively sending a trigger-signal
to a replicated input process on $b$ that represents the receiver-loop’s entry point. In
order to match this protocol of synchronous outputs, the encoding of input-
guarded replication has to check the sender's lock and, based on its value, either
to commit and trigger a copy of its continuation, or to abort the sender.

Evaluation. An encoding is deadlock/divergence-free, if it does not
add deadlocks/divergence loops to the behavior of terms; a deadlock/divergence loop that occurs in (some
derivative of ) an encoded term necessarily results from a deadlock/divergence loop already
occurring in (some derivative of ) the original term. Note that divergence-freedom
implies livelock-freedom.

To prove deadlock-freedom, we take advantage of type information for the
channels that are added in the encoding. We refine channel types according to
Kobayashi’s classification [Kob97], which distinguishes between
reliable and unreliable channels. The following three types of channels are reliable:

$\nu$ linear channels, which are used just once (like our acknowledgement
channels $a$),

$\nu$ broadcast channels, which are used many times (like our acknowledgment
channels $a$),

$\nu$ synchronization channels, which are used exactly once (like our acknowledgment
channels $a$).
Encoding Separate Choice

\[ \mathcal{S}_{\text{sep}} \rightarrow \top \text{ is divergence-free.} \]

Proof: only sketched, but reasonably straightforward.

\[ \mathcal{S}_{\text{sep}} \rightarrow \top \text{ is deadlock-free.} \]

Proof: using a sophisticated type system by Kobayashi et.al.
So, we seem to know how to implement input-guards and how to implement output-guards.

Why not reuse the same encoding also for the case of mixed choice?
Symmetric Cyclic Wait

\[ P | Q \overset{\text{def}}{=} y_0 ![0].P_0 + y_1 ?[x].P_1 \]
\[ | y_0 ?[x].Q_0 + y_1 ![1].Q_1 \]

\[ [ y?[\bar{x}].P ]_l \overset{\text{def}}{=} (\nu b) (b[] | b?[*[]). \]
\[ y?[r, a, \bar{x}]. \]
\[ \text{test } l \]
\[ \text{then test } r \]
\[ \text{then } l![f] | r![f] | a![t] | [ P ] \]
\[ \text{else } l![t] | r![f] | a![f] | b[] \]
\[ \text{else } l![f] | y?[r, a, \bar{x}] \) \]
Incest

\[ I := y!\left[z\right].P + y?\left[x\right].Q. \]

No Intra-Communication possible on the source! But the encoding of separate choice may deadlock ...
Nondeterminism/Randomization

We would need a mixed choice construct. Note that we cannot use internal choice either, because it would only delay potential deadlocks, which arise when the internal decision favors the branch "waiting for the second lock." In Fig. 6, we model a randomized solution based on the encoding in Fig. 5 by only supplying a new clause for receivers.

We use a local state, implemented as a mutex channel \( s \) that carries a tag (and a boolean value) that tells whether none (tag \( N \), w.l.o.g. with value \( f \)), the local (tag \( L \)), or the remote (tag \( R \)) lock are currently held by the receiver. The tag-information, initially \( N \), is supplied by two processes, called lock-checkers, waiting at \( lcl \) and \( rmt \), which try to get hold of the local lock \( l \) and remote lock \( r \), respectively. After grabbing a lock, these processes need to read the current state; if the complementary lock is already held, then the two lock values are passed on to the analyzer process waiting at \( bth \) and the state \( s \) is initialized. Otherwise, the state \( s \) is appropriately updated to announce success for getting the current lock and, in addition to this announcement, a randomizer process at \( rnd \) is started that competes with the lock-checkers for reading the state. If the randomizer succeeds in reading the state, it resets the state and resends the lock, while restarting the corresponding lock-checker. If both lock-checkers succeed reading the state without the randomizer interfering, then \( s \) is left with its initial value and is finally consumed by the active randomizer to terminate the system by resetting the state without restarting any of the lock-checkers and without restarting the randomizer itself. Note that after restarting the whole receiver at \( b \) in the case of local success (\( b_L = t \)) and remote failure (\( b_R = f \)), a new state will be created, when a new request on \( y \) arrives.

Evaluation. As the encoding for separate choice, the randomized encoding for mixed choice in Fig. 6 is uniform since restriction and parallelism are encoded.

\[
\llbracket y[\tilde{x}].P \rrbracket_l \overset{\text{def}}{=} (\nu b)(b[] \mid b^*\cdot y[r, a, \tilde{x}]).
\]

\[
(\nu s, lcl, rmt, rnd, bth) (lcl^*[] \cdot l?[b_L] \cdot s?[tag, b] .
\]

\[
\text{if } tag=\text{R} \text{ then } bth![b_L, b] \mid s![N, f] \text{ else } s![L, b_L] \mid \text{rnd}[]
\]

\[
rmt^*[] \cdot r?[b_R] \cdot s?[tag, b] .
\]

\[
\text{if } tag=\text{L} \text{ then } bth![b, b_R] \mid s![N, f] \text{ else } s![R, b_R] \mid \text{rnd}[]
\]

\[
rnd^*[] \cdot s?[tag, b] . (s![N, f] \mid \text{if } tag=\text{L} \text{ then } l![b] \mid lcl[] \text{ else }
\]

\[
\text{if } tag=\text{R} \text{ then } r![b] \mid rmt[] \text{ else } 0
\]

\[
bth?[b_L, b_R] . \text{ if } b_L \land b_R \text{ then } l![f] \mid r![f] \mid a![t] \mid [P] \text{ else }
\]

\[
\text{if } b_L \text{ then } l![t] \mid r![f] \mid a![f] \mid b[] \text{ else }
\]

\[
\text{if } b_R \text{ then } l![f] \mid r![t] \mid y[r, a, \tilde{x}]
\]

\[
\text{else } l![f] \mid r![f] \mid a![f]
\]

\[
lcl[] \mid rmt[] \mid s![N, f]
\)
Nondeterminism/Randomization

The encoding is:

- loosely inspired by [Rabin, Lehmann 94]
- (strongly) compositional
- obeys a name discipline
- deadlock-free
- ... but not livelock-free ... it introduces divergence
- fully abstract in a very restricted way ...
- a candidate for an explicit probabilistic treatment
Mixed-Guarded Choice

\[
\begin{align*}
[\nu x P] &= \nu x[P] \\
[P_1 | P_2] &= [P_1] | [P_2] \\
[X] &= X \\
[\text{rec}_X P] &= \text{rec}_X[P]
\end{align*}
\]

\[
\begin{align*}
\left[ \sum_i \alpha_i.P_i \right] + \sum_j \tau.Q_j + \sum_k \beta_k.R_k 
\right] &= \nu l (\overline{lt} | \nu h (\overline{lh} | \prod_i [\alpha_i.P_i]_{lh}) | \prod_j [\tau.Q_j]_l | \prod_k [\beta_k.R_k]_l)
\end{align*}
\]

\[
\begin{align*}
[\overline{xy}.P]_{rh} &= \nu a (\overline{x}\langle r, a, h, y \rangle | a(b). \text{if } b \text{ then } [P] \text{ else } 0) \\
[\tau.Q]_l &= l(b). (\overline{lf} | \text{if } b \text{ then } [Q] \text{ else } 0) \\
[x(y).R]_l &= \text{rec}_X (x(r, a, h, y).h.rec_Y (1/2 \tau.l(b_L).((1 - \epsilon) r(b_R).B + \epsilon \tau.(\overline{lb_L} | Y)) + 1/2 \tau.r(b_R).((1 - \epsilon) l(b_L).B + \epsilon \tau.(\overline{rb_R} | Y))))
\end{align*}
\]

where

\[
B = \begin{cases} 
\text{if } b_L \land b_R \text{ then } \overline{h} | \overline{lf} | \overline{ft} | \overline{at} | [R] \\
\text{else if } b_L \text{ then } \overline{h} | \overline{lt} | \overline{ft} | \overline{af} | X \\
\text{else if } b_R \text{ then } \overline{h} | \overline{lf} | \overline{ft} | \overline{x}\langle r, a, h, y \rangle \\
\text{else } \overline{h} | \overline{lf} | \overline{ft} | \overline{af}
\end{cases}
\]

co-stimulated the development of probabilistic Pi Calculus

F.A. w.r.t. may/must testing
The following encoding is **not strongly compositional**! In fact, it is **centralized** ...

The `top-level` adds another layer to the encoding, so it cannot be composed with other instances.
A “Bakery” Algorithm (II)

\[
[y?\tilde{x}.P]_{\text{enc}} \quad \text{def} \quad (\nu b) \ (b![] \mid b?[]).
\]

**equality check**

prevents from incest

(but produces divergence)

\[
y?[m, r, a, \tilde{x}].
\]

if \( n=m \) then \((y?[m, r, a, \tilde{x}] \mid b![])\) else

if \( n<m \)

then test \( l \)

then test \( r \)

then \( l![f] \mid r![f] \mid a![t] \mid [P]_{\text{enc}} \)

else \( l![t] \mid r![f] \mid a![f] \mid b![] \)

else \( l![f] \mid y?[m, r, a, \tilde{x}] \)

else test \( r \)

then test \( l \)

then \( l![f] \mid r![f] \mid a![t] \mid [P]_{\text{enc}} \)

else \( l![f] \mid r![t] \mid y?[m, r, a, \tilde{x}] \)

else \( r![f] \mid a![f] \mid b![]\)
Overview

Part 0a: Encodings vs Full Abstraction
Comparison of Languages/Calculi
Correctness of Encodings

Part 0b: Asynchronous Pi Calculus

Part 1: Input-Guarded Choice
Encoding (Distributed Implementation)
Decoding (Correctness Proof)

Part 2: Output-Guarded Choice
Encoding Separate Choice
Encoding Mixed Choice

Conclusions
A bit of cheating: encodings depend on n-ary choice.

The role of name-passing.

Asynchronous Pi is “sufficiently” expressive ... ... for programming. (Who needs mixed choice?)

The role of coupled simulation.

There is no theory of encodings yet.

The relation to distributed implementations —and to Distributed Computing in general—is not yet sufficiently understood.