Process Algebras and Concurrent Systems

IMT- Institutions Markets Technologies - Alti Studi Lucca

JOIN CALCULUS

Roberto Bruni
Dipartimento di Informatica
Università di Pisa

Contents

- Introduction
- Join calculus + Examples
- Join and π

Contents

- Introduction
- Join calculus + Examples
- Join and π

Join calculus vs. π calculus

- Join is essentially π with restrictions on communication patterns
  - Join combines restriction, reception and replication in a single receptor definition
    - not available separately
  - asynchronous calculus, continuation passing style
    - asynchrony forces one to create and send continuations
- Nevertheless, join and (asynchronous) π have the same expressive power
  - demonstrated by fully abstract encodings in each direction
    - (up to weak barbed congruence)

Motivation (back to 1995)

- Join calculus has been devised to bridge the gap between mathematical abstractions and distributed programming languages
  - process calculus presentation
  - basis for a practical programming language design

Join Calculus vs Petri Nets

- Join-calculus can be seen also as the natural higher order extension of Petri nets
  - places as ports / channels
  - tokens carry values
  - names of places are also admissible values
  - firing can generate fresh pieces of nets
    - new places
    - new transitions

Contents

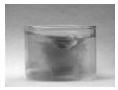
- Introduction
- Join calculus + Examples
- Join and π

Chemical Abstract Machine

- States are called solutions s
  - Multisets of molecules $m_1,\ldots,m_n$
    - data and rules (reflexive CHAM)
- Evolution (chemical rules)

Chemical Abstract Machine

- States are called solutions s
  - Multisets of molecules $m_1,\ldots,m_n$
    - data and rules (reflexive CHAM)
- Evolution (chemical rules)
  - Heating / cooling $\leftrightarrows$ (reversible)
    - Structural equivalence

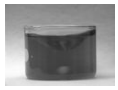


Chemical Abstract Machine

- States are called solutions s
  - Multisets of molecules $m_1,\ldots,m_n$
    - data and rules (reflexive CHAM)
- Evolution (chemical rules)
  - Heating / cooling $\leftrightarrows$ (reversible)
    - Structural equivalence
  - Reactions $\rightarrow$ (not reversible)
    - Transitions

Chemical Abstract Machine

- States are called solutions s
  - Multisets of molecules $m_1,\ldots,m_n$
    - data and rules (reflexive CHAM)
- Evolution (chemical rules)
  - Heating / cooling $\leftrightarrows$ (reversible)
    - Structural equivalence
  - Reactions $\rightarrow$ (not reversible)
    - Transitions



Example: Cell Abstraction

$$\mathrm{get}\langle k\rangle \mid s\langle v\rangle \triangleright k\langle v\rangle \mid s\langle v\rangle$$

- A cell s contains the value v
- To get the value:
  - send a message on port get
  - the parameter k is the return address, where the value v will be sent to

Example: Cell Abstraction

$$\begin{aligned} &get\langle k\rangle \mid s\langle v\rangle \triangleright k\langle v\rangle \mid s\langle v\rangle \\ &set\langle m,k\rangle \mid s\langle v\rangle \triangleright k\langle\rangle \mid s\langle m\rangle \end{aligned}$$

- A cell s contains the value v
- To set the value:
  - send a message on port set
  - the parameter m is the new value for s
  - k is the return address (for confirmation)

Example: Cell Abstraction

$$\begin{aligned} \textbf{def}\quad & get\langle k\rangle \mid s\langle v\rangle \triangleright k\langle v\rangle \mid s\langle v\rangle \\ \wedge\ & set\langle m,k\rangle \mid s\langle v\rangle \triangleright k\langle\rangle \mid s\langle m\rangle \\ \textbf{in}\quad & s\langle n\rangle \end{aligned}$$

- The initial value in s is n
- But get, set and s are locally bound by **def**
  - get and set must be extruded, otherwise no one can use them
  - instead, s is kept private (encapsulation)

Example: Cell Abstraction

$$\begin{aligned} \textbf{def}\quad & get\langle k\rangle \mid s\langle v\rangle \triangleright k\langle v\rangle \mid s\langle v\rangle \\ \wedge\ & set\langle m,k\rangle \mid s\langle v\rangle \triangleright k\langle\rangle \mid s\langle m\rangle \\ \textbf{in}\quad & s\langle n\rangle \mid c\langle get,set\rangle \end{aligned}$$

- get, set are extruded on public channel c
- But c should be known only by the owner of the cell…

Example: Cell Abstraction

$$\textbf{def}\ create\langle n,c\rangle \triangleright \left(\begin{array}{ll} \textbf{def} & get\langle k\rangle \mid s\langle v\rangle \triangleright k\langle v\rangle \mid s\langle v\rangle \\ \wedge & set\langle m,k\rangle \mid s\langle v\rangle \triangleright k\langle\rangle \mid s\langle m\rangle \\ \textbf{in} & s\langle n\rangle \mid c\langle get,set\rangle \end{array}\right) \textbf{in}\ \ldots$$

- A message to create triggers the outermost def:
  - Three fresh names for s, get and set are allocated
    - the initial value of s is the first parameter n
    - get and set are sent back to the second argument c
    - instead s will never be extruded
  - Invariant
    - in every configuration there is exactly one message on s

Join Calculus in One Slide

- Syntax

processes

definitions

patterns

P,Q	::=	0 | $x\langle\hat{y}\rangle$ | **def** D **in** P | P|Q	
D,E	::=	J▷P | D∧E	resembles funct. prog. let f(x)=E in F (same scoping discipline)
J,K	::=	$x\langle\hat{y}\rangle$ | J|K	

- Operational semantics (CHAM Style)

heating and cooling

reaction

0		⇆	
P|Q		⇆	P,Q
D∧E		⇆	D,E
def D **in** P		⇆	$D\sigma_{dn(D)}$, $P\sigma_{dn(D)}$ (range $\sigma_{dn(D)}$ "globally fresh")
J▷P, Jσ		→	J▷P, Pσ

JOIN: An Example

- A process P
  - P ≡ $z\langle x,z\rangle$ | **def** $x\langle y\rangle \triangleright z\langle y,x\rangle$ **in** $x\langle v\rangle$

defined name

free name

free name

received name

bound name

bound name

- P as a solution
  - $z\langle x,z\rangle$, $w\langle y\rangle \triangleright z\langle y,w\rangle$, $w\langle v\rangle$
- A reaction
  - $z\langle x,z\rangle$, $w\langle y\rangle \triangleright z\langle y,w\rangle$, $w\langle v\rangle \rightarrow$
    - $z\langle x,z\rangle$, $w\langle y\rangle \triangleright z\langle y,w\rangle$, $z\langle v,w\rangle$

extrusion

HOMEWORK

- Guess the meaning of:
  - **def** x⟨u⟩ ▷ y⟨u⟩ **in** P
  - **def** y⟨u⟩ ▷ x⟨u⟩ **in def** x⟨u⟩ ▷ y⟨u⟩ **in** P
  - **def** s⟨⟩ ▷ P ∧ s⟨⟩ ▷ Q **in** s⟨⟩
  - **def** (c⟨⟩ ▷ P|c⟨⟩) **in** Q|c⟨⟩

Example: Mailbox (sketched)

rb@gmail.it⟨from,subj,msg⟩ | save⟨⟩ ▷ store⟨from,subj,msg⟩ | save⟨⟩

chfwd⟨email⟩ | save⟨⟩ ▷ fwd⟨email⟩
rb@gmail.it⟨from,subj,msg⟩ | fwd⟨email⟩ ▷
store⟨from,subj,msg⟩ | email⟨from,subj,msg⟩ | fwd⟨email⟩

chvacation⟨info⟩ | save⟨⟩ ▷ vacation⟨info⟩
rb@gmail.it⟨from,subj,msg⟩ | vacation⟨info⟩ ▷
store⟨from,subj,msg⟩ | from⟨rb@gmail.it,"Away",info⟩ | vacation⟨info⟩

inbox⟨get,next⟩ | store⟨from,subj,msg⟩ ▷
def elget⟨k⟩ | elem⟨f,s,m,g,n⟩ ▷ k⟨f,s,m⟩ | elem⟨f,s,m,t,n⟩
∧ elnext⟨k⟩ | elem⟨f,s,m,g,n⟩ ▷ k⟨g,n⟩
in inbox⟨elget,elnext⟩ | elem⟨from,subj,msg,get,next⟩

Continuation Passing Style I

- The form of definitions resembles very much
  - **let** f(x)=E **in** E′ (typical of functional programming)
    - e.g. same scoping discipline
- Asynchrony forces us to create and send continuations in join
  - e.g. encoding untyped λ-calculus
    - $[M]_v$ sends the value of M on v
    - a value is a process serving requests
    - a request must supply two names
      - x (channel for requests for the value of the argument)
      - w (to eventually return a value)

Continuation Passing Style II

- Call-by-name
  - $[x]_v$ = x⟨v⟩
  - $[\lambda x.M]_v$ = **def** k⟨x,w⟩ ▷ $[M]_w$ **in** v⟨k⟩
  - $[MN]_v$ = **def** y⟨p⟩ ▷ $[N]_p$
  - **in def** q⟨c⟩ ▷ c⟨y,v⟩ **in** $[M]_q$
- Parallel call-by-value
  - $[x]_v$ = v⟨x⟩
  - $[\lambda x.M]_v$ = **def** k⟨x,w⟩ ▷ $[M]_w$ **in** v⟨k⟩
  - $[MN]_v$ = **def** q⟨c⟩|p⟨y⟩ ▷ c⟨y,v⟩ **in** $[M]_q|[N]_p$

Call-by-Name

- Strategy: leftmost order, no reduction under λ
- Reductions are entirely sequential
- The image of the translation is exactly the deterministic subset of Join (no parallel composition, no conjunction)
  - $[x]_v$ = x⟨v⟩
  - $[\lambda x.M]_v$ = **def** k⟨x,w⟩ ▷ $[M]_w$ **in** v⟨k⟩
  - $[MN]_v$ = **def** y⟨p⟩ ▷ $[N]_p$
  - **in def** q⟨c⟩ ▷ c⟨y,v⟩ **in** $[M]_q$

Call-by-Name: Example

- $[(\lambda x.M)N]_v$ = **def** y⟨p⟩ ▷ $[N]_p$
  in def q⟨c⟩ ▷ c⟨y,v⟩ **in** $[\lambda x.M]_q$
- $[(\lambda x.M)N]_v$ = **def** y⟨p⟩ ▷ $[N]_p$
  in def q⟨c⟩ ▷ c⟨y,v⟩
  in def k⟨x,w⟩ ▷ $[M]_w$ **in** q⟨k⟩
- $[(\lambda x.M)N]_v \rightarrow$ **def** y⟨p⟩ ▷ $[N]_p$
  in def q⟨c⟩ ▷ c⟨y,v⟩
  in def k⟨x,w⟩ ▷ $[M]_w$ **in** k⟨y,v⟩
- $[(\lambda x.M)N]_v \rightarrow^*$ **def** y⟨p⟩ ▷ $[N]_p$
  in def q⟨c⟩ ▷ c⟨y,v⟩
  in def k⟨x,w⟩ ▷ $[M]_w$ **in** $[M[y/x]]_v$

Call-by-Name: Example

- $[(\lambda x.M)N]_v$ = **def** $y\langle p\rangle \triangleright [N]_p$ **in def** $q\langle c\rangle \triangleright c\langle y,v\rangle$ **in** $[\lambda x.M]_q$
- $[(\lambda x.M)N]_v$ = **def** $y\langle p\rangle \triangleright [N]_p$ **in def** $q\langle c\rangle \triangleright c\langle y,v\rangle$ **in def** $k\langle x,w\rangle \triangleright [M]_w$ **in** $q\langle k\rangle$
- $[(\lambda x.M)N]_v \rightarrow$ **def** $y\langle p\rangle \triangleright [N]_p$ **in def** $q\langle c\rangle \triangleright c\langle y,v\rangle$ **in def** $k\langle x,w\rangle \triangleright [M]_w$ **in** $k\langle y,v\rangle$
- $[(\lambda x.M)N]_v \rightarrow^*$ **def** $y\langle p\rangle \triangleright [N]_p$ **in def** $q\langle c\rangle \triangleright c\langle y,v\rangle$ **in def** $k\langle x,w\rangle \triangleright [M]_w$ **in** $[M[y/x]]_v$

Call-by-Name: Example

- $[(\lambda x.M)N]_v$ = **def** $y\langle p\rangle \triangleright [N]_p$ **in def** $q\langle c\rangle \triangleright c\langle y,v\rangle$ **in** $[\lambda x.M]_q$
- $[(\lambda x.M)N]_v$ = **def** $y\langle p\rangle \triangleright [N]_p$ **in def** $q\langle c\rangle \triangleright c\langle y,v\rangle$ **in def** $k\langle x,w\rangle \triangleright [M]_w$ **in** $q\langle k\rangle$
- $[(\lambda x.M)N]_v \rightarrow$ **def** $y\langle p\rangle \triangleright [N]_p$ **in def** $q\langle c\rangle \triangleright c\langle y,v\rangle$ **in def** $k\langle x,w\rangle \triangleright [M]_w$ **in** $k\langle y,v\rangle$
- $[(\lambda x.M)N]_v \rightarrow^*$ **def** $x\langle p\rangle \triangleright [N]_p$ **in** $[M]_v$

Parallel Call-by-Value

- Strategy: again no reduction under λ, but in (TU), T and U can be evaluated in parallel
- Confluent, but non deterministic
  - $[x]_v = v\langle x\rangle$
  - $[\lambda x.M]_v$ = **def** $k\langle x,w\rangle \triangleright [M]_w$ **in** $v\langle k\rangle$
  - $[MN]_v$ = **def** $q\langle c\rangle | p\langle y\rangle \triangleright c\langle y,v\rangle$ **in** $[M]_q | [N]_p$

Call-by-Value: Example

- $[(\lambda x.M)N]_v$ = **def** $q\langle c\rangle | p\langle y\rangle \triangleright c\langle y,v\rangle$ **in** $[\lambda x.M]_q \mid [N]_p$
- $[(\lambda x.M)N]_v$ = **def** $q\langle c\rangle | p\langle y\rangle \triangleright c\langle y,v\rangle$ **in** $[N]_p \mid$ **def** $k\langle x,w\rangle \triangleright [M]_w$ **in** $q\langle k\rangle$
- $[(\lambda x.M)N]_v \rightarrow^*$ **def** $q\langle c\rangle | p\langle y\rangle \triangleright c\langle y,v\rangle$ **in** $p\langle z\rangle \mid$ **def** $k\langle x,w\rangle \triangleright [M]_w$ **in** $q\langle k\rangle$
- $[(\lambda x.M)N]_v \rightarrow$ **def** $q\langle c\rangle | p\langle y\rangle \triangleright c\langle y,v\rangle$ **in def** $k\langle x,w\rangle \triangleright [M]_w$ **in** $k\langle z,v\rangle$
- $[(\lambda x.M)N]_v \rightarrow$ **def** $q\langle c\rangle | p\langle y\rangle \triangleright c\langle y,v\rangle$ **in def** $k\langle x,w\rangle \triangleright [M]_w$ **in** $[M[z/x]]_v$

Contents

- Introduction
- Join calculus + Examples
- Join and π

Core Join Calculus

- Syntax
  - a unique syntactic category

    P,Q ::= $x\langle u\rangle$ | **def** $x\langle u\rangle | y\langle v\rangle \triangleright$ Q **in** P | P|Q
- Operational semantics
  - CHAM Style
  - (but also LTS is defined)
- The core join calculus has the same expressive power as the full join-calculus
  - via a fully-abstract encoding

Full abstraction

- Two process calculi with equivalences $\approx_1$ and $\approx_2$
- The first is more expressive than the second if we can find a fully abstract encoding $[.]_{2\to1}$
  - i.e. an encoding such that
  - $P \approx_2 Q$ iff $[P]_{2\to1} \approx_1 [Q]_{2\to1}$
- The two calculi have the same expressive power if each one is more expressive than the other
  - (If one is a sub-calculus of the other, then one implication is obvious)

What is Observable?

- Communication
  - on internal names (no)
  - on free names (yes)
- Internal steps
  - countable: strong semantics (no)
  - immaterial: weak semantics (yes)
- Equivalence
  - reflexive, symmetric and transitive (yes)
  - closed under contexts: congruence (yes)

Basic Observations

- Processes interact with the outside
  - by extruding names on free ports
  - by waiting for answers (via enclosed definitions)
- Processes are distinguished on the basis of their ability to emit messages on their free ports
  - weak asynchronous output barb $\Downarrow_x$
  - $P\Downarrow_x$ iff
    - x is a free name in P
    - and $\exists u$ such that $P \to^* Q, x\langle u\rangle$

Remarks on Barbs

- Two processes P and Q such that
  - $\exists u$ with $P\Downarrow_x$ but $\neg(Q\Downarrow_x)$
- cannot be reasonably identified!
- Barbs are just elementary experiments
  - barbs do not count reductions (ok)
  - barbs do not observe branching (uhm)
  - barbs do not observe message reception (uhm)

Closure Under Reductions

- Reductions are mute transitions
  - i.e. only trivial labels are present
    - $P\to P'$ can be read as $P\text{—}\tau\to P'$
- In ordinary (strong) bisimulation
  - if $P\approx Q$ and $P\to P'$, then $\exists Q'\approx P'$ s.t. $Q\to Q'$
  - (and vice versa)
- In weak bisimulation
  - if $P\approx Q$ and $P \to^* P'$, then $\exists Q'\approx P'$ s.t. $Q \to^* Q'$
  - (and vice versa)

Closure Under Contexts

- If $P\approx Q$ we expect that P and Q can be used interchangeably in any larger process
  - but $P \equiv a\langle b\rangle$ and $Q \equiv a\langle c\rangle$ look equivalent when taken in isolation
    - no reduction, a unique barb $\Downarrow_a$
  - however, they are not equivalent in the context
    - **def** $a\langle x\rangle \triangleright x\langle\rangle$ **in** [.]
  - as in fact
    - **def** $a\langle x\rangle \triangleright x\langle\rangle$ **in** $P \to b\langle\rangle$ (i.e. **def** $a\langle x\rangle \triangleright x\langle\rangle$ **in** $P \Downarrow_b$)
    - **def** $a\langle x\rangle \triangleright x\langle\rangle$ **in** $Q \to c\langle\rangle$ (i.e. **def** $a\langle x\rangle \triangleright x\langle\rangle$ **in** $Q \Downarrow_c$)

The Observational Congruence

- We take the largest equivalence relation $\approx$ that
  - is a refinement of output barbs
    - if $P \approx Q$ then $(\forall x.\ P{\Downarrow_x}$ iff $Q{\Downarrow_x})$
  - is closed under weak reduction
    - if $P \approx Q$ and $P \rightarrow^* P'$, then $\exists Q' \approx P'$ s.t. $Q \rightarrow^* Q'$
  - is a congruence w.r.t. definitions and parallel
    - if $P \approx Q$ then $(\forall D.$ **def** D **in** $P \approx$ **def** D **in** $Q)$
    - if $P \approx Q$ then $(\forall R.\ P|R \approx Q|R)$

Observational Congruence: Examples

- If $fn(P)=\varnothing$ then $P \approx 0$
- If $P \equiv Q$ then $P \approx Q$
- $a\langle u\rangle \not\approx b\langle u\rangle$
- $a\langle b\rangle \not\approx a\langle c\rangle$
- $a\langle b\rangle \approx$ **def** $c\langle x\rangle \triangleright b\langle x\rangle$ **in** $a\langle c\rangle$
  - it is not possible to distinguish between different names that exhibit the same external behaviour

Core Join vs Full Join

- Expressiveness-preserving simplification of syntax
  - recursive binding
    - shift binding variables from definition to reception
      - **def** $J \triangleright Q$ **in** P becomes **def** $J|b\langle\tilde{a},b'\rangle \triangleright Q|b\langle\tilde{a},b'\rangle$ **in** $P|b\langle\tilde{a},b\rangle$
      - where $\tilde{a}$ is the vector of variables in $fn(Q)\cap dn(J)$
  - complex definitions
    - n-way join patterns and multiple clauses connected by $\wedge$ as sequences joining two atoms at most
  - polyadic messages
    - name tuples are communicated by using auxiliary private names

Asynchronous π

- Syntax — processes

  P,Q ::= $x\langle u\rangle \mid x(u).P \mid \nu u.P \mid !x(u).P \mid P|Q$
- Abstract semantics
  - asynchronous barbed congruence
  - ex. $x(u).x\langle u\rangle \approx 0$
  - ex. equator $EQ(x,y) \equiv !x(u).y\langle u\rangle \mid !y(v).x\langle v\rangle$
    - $P\{x/y\} \approx Q\{x/y\}$ implies $EQ(x,y)|P \approx EQ(x,y)|Q$

Naïve Encoding: Join in π

- $[x\langle v\rangle]_{j\rightarrow\pi} = x\langle v\rangle$
- $[P|Q]_{j\rightarrow\pi} = [P]_{j\rightarrow\pi} \mid [Q]_{j\rightarrow\pi}$
- $[$**def** $x\langle u\rangle|y\langle v\rangle \triangleright Q$ **in** $P]_{j\rightarrow\pi} = \nu x.\nu y.(!x(u).y(v).\ [Q]_{j\rightarrow\pi} \mid [P]_{j\rightarrow\pi})$
- In the translation we loose
  - the symmetry between x and y
  - the atomicity of their joint reduction
    - it does not matter, because x and y are restricted
- Not closed under π contexts!!!
  - if x or y are extruded, then new receptors could appear

Problems with Full Abstraction of Join in π: Example

- Let $P \equiv [$ **def** $x\langle\rangle \triangleright 0$ **in** $a\langle x\rangle \mid x\langle\rangle\ ]_{j\rightarrow\pi}$
- and $Q \equiv [$ **def** $y\langle\rangle \triangleright 0$ **in** $a\langle y\rangle\ ]_{j\rightarrow\pi}$
  - the two encoded processes are equivalent
  - P and Q are not
- Take the π-context $\nu a.(\ a(u).u().b\langle\rangle \mid [.]\ )$
  - then $\nu a.(\ a(u).u().b\langle\rangle \mid P\ ) \Downarrow_b$
  - while $\neg(\ \nu a.(\ a(u).u().b\langle\rangle \mid Q\ ) \Downarrow_b\ )$

Naïve Encoding: π in Join

- Each π-channel x is simulated by two ports
  - x_o for output (where emitters send values)
  - x_i for input (the receiver defines a name k for its continuation and sends it as a reception offer on x_i)
  - $[x\langle v\rangle]_{\pi\to j} = x_o\langle v_o, v_i\rangle$
  - $[x(u).P]_{\pi\to j} = \mathbf{def}\ k\langle v_o, v_i\rangle \triangleright [P]_{\pi\to j}\ \mathbf{in}\ x_i\langle k\rangle$
  - $[\nu x.P]_{\pi\to j} = \mathbf{def}\ x_o\langle v_o, v_i\rangle \mid x_i\langle k\rangle \triangleright k\langle v_o, v_i\rangle\ \mathbf{in}\ [P]_{\pi\to j}$
  - $[!x(u).P]_{\pi\to j} = \mathbf{def}\ k\langle v_o, v_i\rangle \triangleright x_i\langle k\rangle \mid [P]_{\pi\to j}\ \mathbf{in}\ x_i\langle k\rangle$
  - $[P|Q]_{\pi\to j} = [P]_{\pi\to j} \mid [Q]_{\pi\to j}$
- Not closed under Join contexts!!!
  - problems with free names and input barbs

Problems with Full Abstraction of π in Join: Examples

- $[\ x\langle a\rangle \mid x\langle b\rangle \mid x(u).y\langle u\rangle\ ]_{\pi\to j}$
  - cannot reduce because there is no englobing νx
- $[\ x(u).x\langle u\rangle\ ]_{\pi\to j}$
  - exhibits a barb on x_i that reveals the presence of an input on x
- A context could provide messages with arbitrary i/o-pairs
  - ex. mismatched order: $x_o\langle v_i, v_o\rangle$
  - ex. mismatched names: $x_o\langle v_o, w_i\rangle$

Features (as process calculus)

- Based on an elementary model of concurrency
  - reflexive chemical abstract machine = generic CHAM + imposing locality + adding reflection
    - locality: only linear reaction patterns allowed
      - each molecule or reaction rule is associated to a single reaction site
      - more effective than generic CHAM: molecules travel to their reaction site, instead of having to blindly mix and match
    - reflection: reactions can generate new kinds of molecules together with their defining reaction rules
      - computational completeness of the model
- λ-calculus as sequential deterministic subset

Features (as distributed programming language)

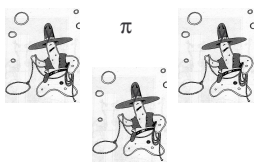
- Extends a higher-order functional language
  - parallelism in expressions (fork calls)
  - parallelism in function patterns (join patterns)
    - jointly defined function provide the same capabilities as synchronous channels or concurrent objects
    - join patterns are more consistent with lexical scope
      - static binding of function calls to the code
      - as opposed to dynamic binding of messages to receptors
- Distributed implementations
  - JoCaml, JoinJava, Polyphonic C#

Implementability

- Uniqueness of receptors favour distributed implementation of Join

Join

π



Polyphonic C# (Cω)

- Methods can be
  - synchronous
    - (caller is blocked)
  - asynchronous
    - (no result, caller can proceed almost immediately)
- Key feature
  - The same body (called a chord) can be associated with a set of asynchronous and (at most one) synchronous methods
  - A method can appear in the header of several chords
  - The body is executed only if all methods in its header have been called

Unbounded Size Buffer

```
public class Buffer {

   public String get()
   & public async put(String s){
       return s;
   }

}
```

synchronous

asynchronous

chord

One-Place Buffer

```
public class OnePlaceBuffer {
   public OnePlaceBuffer() {
      empty();
   }
   public void put(String s)
   & private async empty() {
      contains(s);
      return;
   }
   public String get()
   & private async contains(String s) {
      empty();
      return s;
   }
}
```


Reader-Writer Lock

```
class ReaderWriter { ReaderWriter()         {  Idle();  }
                     void Exclusive()
                     & private async Idle()  {          }

                     void ReleaseExclusive()  {  Idle();  }

                     void Shared()
                     & private async Idle()   {  S(1);    }

                     void Shared()
                     & private async S(int n) {  S(n+1);  }

                     void ReleaseShared()
                     & private async S(int n) {
                        if (n == 1) Idle(); else S(n-1);
                     }                                    }
```


References

- The reflexive chemical abstract machine and the Join calculus (Proc. POPL'96, ACM, pp. 372-385)
  - C. Fournet, G. Gonthier