Flat Committed Join in Join

Hernán Melgratti
joint work with Roberto Bruni and Ugo Montanari
Dipartimento di Informatica - Università di Pisa

Committed Join (cJoin)
- Join + primitives for negotiations
- Syntax:

Messages

Processes:
P, Q := 0 | x[y]| M N

Definitions:
D := \text{def} D \text{ in } P_1 \text{ or } P_2

Patterns:
\lambda x \text{ .. } | \lambda y \text{ .. }

Committed Join (cJoin)
- Join + primitives for negotiations
- Syntax:

Messages

Processes:
P, Q := 0 | x[y]| M N

Definitions:
D := \text{def} D \text{ in } P_1 \text{ or } P_2

Patterns:
\lambda x \text{ .. } | \lambda y \text{ .. }

Committed Join (cJoin)
- Join + primitives for negotiations
- Syntax:

Messages

Processes:
P, Q := 0 | x[y]| M N

Definitions:
D := \text{def} D \text{ in } P_1 \text{ or } P_2

Patterns:
\lambda x \text{ .. } | \lambda y \text{ .. }

Committed Join (cJoin)
- Join + primitives for negotiations
- Syntax:

Messages

Processes:
P, Q := 0 | x[y]| M N

Definitions:
D := \text{def} D \text{ in } P_1 \text{ or } P_2

Patterns:
\lambda x \text{ .. } | \lambda y \text{ .. }

Committed Join (cJoin)
- Operational Semantics (CHAM Style):

0 =
P Q = P Q
D E = D E
\text{def } D \text{ in } P = D E\text{ or } P Q
\lambda x \text{ .. } \lambda y \text{ .. } \text{merge \text{ .. }}

Merge definition
Programmable abort
Negotiation
Compensation

heating and cooling
reaction

Workshop Fiore Coimbra – Dec, 2023

Workshop Fiore Coimbra – Dec, 2023

Workshop Fiore Coimbra – Dec, 2023

Workshop Fiore Coimbra – Dec, 2023

Workshop Fiore Coimbra – Dec, 2023
Committed Join (cJoin)

- Operational Semantics (CHAM Style):
  \[ 0 \triangleq P \\
  P/Q \triangleq P.Q \\
  D/E \triangleq D.E \\
  \text{def } D \text{ in } P \triangleq \text{Def}_{P/Q} \cup \text{Def}_{D/E} \text{ range(c) fresh} \\
  3 \cdot P, 3e \rightarrow 3 \cdot P, Pe \\
  [P;Q] \triangleq ([P, Q]) \]

Compensation Q is kept frozen

Example: Mailing List

ML = MailingList(k) > MLDef

MLDef = def ...

in k(add, tell, close) | List

Example: Mailing List

ML = MailingList(k) > MLDef

MLDef = def ...

in k(add, tell, close) | List

Committed Join (cJoin)

- Operational Semantics (CHAM Style):
  \[ 0 \triangleq P \\
  P/Q \triangleq P.Q \\
  D/E \triangleq D.E \\
  \text{def } D \text{ in } P \triangleq \text{Def}_{P/Q} \cup \text{Def}_{D/E} \text{ range(c) fresh} \\
  3 \cdot P, 3e \rightarrow 3 \cdot P, Pe \\
  [P;Q] \triangleq ([P, Q]) \]

Global Resources: Commit

Committed Join (cJoin)

- Operational Semantics (CHAM Style):
  \[ 0 \triangleq P \\
  P/Q \triangleq P.Q \\
  D/E \triangleq D.E \\
  \text{def } D \text{ in } P \triangleq \text{Def}_{P/Q} \cup \text{Def}_{D/E} \text{ range(c) fresh} \\
  3 \cdot P, 3e \rightarrow 3 \cdot P, Pe \\
  [P;Q] \triangleq ([P, Q]) \]

Global Resources: Commit

Committed Join (cJoin)

- Operational Semantics (CHAM Style):
  \[ 0 \triangleq P \\
  P/Q \triangleq P.Q \\
  D/E \triangleq D.E \\
  \text{def } D \text{ in } P \triangleq \text{Def}_{P/Q} \cup \text{Def}_{D/E} \text{ range(c) fresh} \\
  3 \cdot P, 3e \rightarrow 3 \cdot P, Pe \\
  [P;Q] \triangleq ([P, Q]) \]

Global Resources: Commit
Example: Mailing List

ML = MailingList(k) = MLDef

MLDef = def ...

\[ \text{lsty} | \text{add}(y) \triangleright \ldots \]
\[ \text{lsty} | \text{tell}(v) \triangleright \ldots \]
\[ \text{lsty} | \text{close}() \triangleright \ldots \]
\[ \text{in k(add, tell, close) | list-nil} \]

Example: Mailing List

ML = MailingList(k) = MLDef

MLDef = def ...

\[ \text{lsty} | \text{add}(y) \triangleright \text{def} z(v, w) \triangleright x(y) | y(v, w) \text{in listz} \]
\[ \text{lsty} | \text{tell}(v) \triangleright \ldots \]
\[ \text{lsty} | \text{close}() \triangleright \ldots \]
\[ \text{in k(add, tell, close) | list-nil} \]

Example: Mailing List

ML = MailingList(k) = MLDef

MLDef = def ...

\[ \text{lsty} | \text{add}(y) \triangleright \text{def} z(v, w) \triangleright x(y) | y(v, w) \text{in listz} \]
\[ \text{lsty} | \text{tell}(v) \triangleright \text{def} w() \triangleright 0 \text{in y(v, w) listy} | z(lsty) \]
\[ \text{lsty} | \text{close}() \triangleright \ldots \]
\[ \text{in k(add, tell, close) | list-nil} \]

Example: Mailing List

ML = MailingList(k) = MLDef

MLDef = def nil(v, w) \triangleright 0

\[ \text{lsty} | \text{add}(y) \triangleright \text{def} z(v, w) \triangleright x(y) | y(v, w) \text{in listz} \]
\[ \text{lsty} | \text{tell}(v) \triangleright \text{def} w() \triangleright 0 \text{in y(v, w) listy} | z(lsty) \]
\[ \text{lsty} | \text{close}() \triangleright \ldots \]
\[ \text{in k(add, tell, close) | list-nil} \]

Example: Mailing List

ML = MailingList(k) = MLDef

MLDef = def nil(v, w) \triangleright 0

\[ \text{lsty} | \text{add}(y) \triangleright \text{def} z(v, w) \triangleright x(y) | y(v, w) \text{in listz} \]
\[ \text{lsty} | \text{tell}(v) \triangleright \text{def} w() \triangleright 0 \text{in y(v, w) listy} | z(lsty) \]
\[ \text{lsty} | \text{close}() \triangleright \ldots \]
\[ \text{in k(add, tell, close) | list-nil} \]

Implementing flat cJoin

- **Goal**: To implement cJoin in Join
- **Key Points**
  - Distributed commit of interacting negotiations as a global decision
  - Participants may join dynamically (participants statically unknown)
- **First Step**
  - Consider flat negotiations
  - Use canonical form of processes
  - Encode canonical flat cJoin processes asJoin processes
Flat cJoin

- Negotiations cannot be nested
- Type system for cJoin Processes:
  - P: □_0 P does not contain [□_1] at all
  - P: □_1 P may contain [□_1] in the definitions
  - P: □_2 P may have and generate flat negotiations
- Join Processes have type □_2
- Subject Reduction holds for □_0 and □_1
- Flat cJoin: The sub-calculus of all P:□_2

Canonical Flat cJoin

- Inspired by the basic shapes of ZS nets
- Few elementary definition patterns
  - Open \( \alpha_0 \vdash \alpha \) & P: □_1 & \text{count}(P) = 1
  - Ord-Mov \( \alpha_0 \vdash \alpha \) & P: □_1 & \text{count}(P) \leq 2
  - Merge-Mov \( \alpha_0 \vdash \alpha \) & P: □_1 & \text{count}(P) \leq 2
  - Ord-Join \( \alpha_0 \vdash \alpha \) & P: □_1 & \text{count}(P) = 1
  - Merge-Join \( \alpha_0 \vdash \alpha \) & P: □_1 & \text{count}(P) = 1
- Any flat process can be written in canonical form

Canonical Form: Example

\[
\begin{align*}
def \text{nil}(v, w) & \rightarrow 0 \\
\land \text{lst}(y) & \rightarrow \text{add}(x) \rightarrow \text{def} \text{z}(w, x) \rightarrow y \rightarrow \text{lst}(z) \\
\land \text{lst}(y) & \rightarrow \text{tell}(v) \rightarrow \text{def} w, v \rightarrow 0 \rightarrow \text{lst}(y) : \text{lst}(y) \\
\land \text{lst}(y) & \rightarrow \text{close}() \rightarrow 0 \rightarrow \text{lst}(nil) & \text{Has Type} \quad \square_1 \\
\land \text{z}(y, x) & \rightarrow \text{def} w, v \rightarrow 0 \rightarrow \text{lst}(y) : \text{lst}(y) \\
\land \text{z}(y, x) & \rightarrow \text{def} w, v \rightarrow 0 \rightarrow \text{lst}(y) : \text{lst}(y) & \text{Has count} = 2
\end{align*}
\]

Encoding: Main Idea

- Any message in a negotiation runs in a thread, which is managed by a coordinator
- Coordinators perform a D2PC protocol [BLM2002].
  - A variant of the decentralized 2PC with a finite but unknown number of participants
  - When a participant P is ready to commit it has only a partial knowledge of the whole set of participants
  - Only those who directly cooperated with P
  - To commit P must contact all its neighbors and learn the identity of other participants from them

Encoding: D2PC

- Every participant P acts as coordinator
  - During the transaction P builds its own synchronization set \( I_P \) of cooperating agents
  - When P is ready to commit, P asks readiness to processes in \( I_P \) if empty P is isolated and can commit
  - In doing so, P sends them the set \( I_P \)
  - Other participants will send to P
    - either a successful reply with their own synchronization sets
    - or a failure message
    - (in the case, failure is then propagated)
  - Successful replies are added to \( I_P \)
  - The protocol terminates when \( I_P \) is transitively closed

Encoding a negotiation

\[
[\text{def} z()] \rightarrow 0 \rightarrow z() \rightarrow 0
\]
Encoding a negotiation

A negotiation with one thread

\[ \text{def } z(x) \geq 0 \text{ in } z(y) \geq 0 \]

Compensation

- \text{def } D

Set initial compensation

\[ \text{def } D \text{ in state } (\text{cmp}) \]

Encoding a negotiation

A coordinator to manage \( z \)

\[ \text{def } D \text{ in state } (\text{cmp}) \]

Encoding a negotiation

\[ \text{def } z(x) > 0 \text{ in } z(y) > 0 \]

\[ \text{def } D \text{ in state } (\text{cmp}) \]

Encoding a negotiation

\[ \text{def } z(x) > 0 \text{ in } z(y) > 0 \]

\[ \text{def } D \text{ in state } (\text{cmp}) \]

Encoding a negotiation

\[ \text{def } z(x) > 0 \text{ in } z(y) > 0 \]

\[ \text{def } D \text{ in state } (\text{cmp}) \]

Encoding a negotiation

\[ \text{def } z(x) > 0 \text{ in } z(y) > 0 \]

\[ \text{def } D \text{ in state } (\text{cmp}) \]

Encoding a negotiation

\[ \text{def } z(x) > 0 \text{ in } z(y) > 0 \]

\[ \text{def } D \text{ in state } (\text{cmp}) \]

Encoding a negotiation

\[ \text{def } z(x) > 0 \text{ in } z(y) > 0 \]

\[ \text{def } D \text{ in state } (\text{cmp}) \]

Encoding a negotiation

\[ \text{def } z(x) > 0 \text{ in } z(y) > 0 \]

\[ \text{def } D \text{ in state } (\text{cmp}) \]

Encoding a negotiation

\[ \text{def } z(x) > 0 \text{ in } z(y) > 0 \]

\[ \text{def } D \text{ in state } (\text{cmp}) \]

Encoding a negotiation

\[ \text{def } z(x) > 0 \text{ in } z(y) > 0 \]

\[ \text{def } D \text{ in state } (\text{cmp}) \]

Encoding a negotiation

\[ \text{def } z(x) > 0 \text{ in } z(y) > 0 \]

\[ \text{def } D \text{ in state } (\text{cmp}) \]

Encoding a negotiation

\[ \text{def } z(x) > 0 \text{ in } z(y) > 0 \]

\[ \text{def } D \text{ in state } (\text{cmp}) \]

Encoding a negotiation

\[ \text{def } z(x) > 0 \text{ in } z(y) > 0 \]

\[ \text{def } D \text{ in state } (\text{cmp}) \]
**Encoding a negotiation**

```
(def z) > 0 in z(); 0
It consumes managed msgs
```

```
(def D, cmp) > 0
in state [<cmp>]
```

```
D carries the ports of its coordinator
```

```
def D, a, b >
in zput, ab, {lock}
```

**Encoding a negotiation**

```
(def z) > 0 in z(); 0
It consumes managed msgs
```

```
(def D, cmp) > 0
in state [<cmp>]
```

```
D carries the ports of its coordinator
```

```
def D, a, b >
in zput, ab, {lock}
```

**Encoding a negotiation**

```
(def z) > 0 in z(); 0
The thread ends
```

```
(def D, cmp) > 0
in state [<cmp>]
```

```
D can commit
```

```
def D, a, b > p(x, y, z)
in zput, ab, {lock}
```

**Encoding an abort**

```
(def z) = abort in z(); 0
The thread aborts
```

```
(def D, cmp) > 0
in state [<cmp>]
```

```
D aborts
```

```
def D, a, b > a
in zput, ab, {lock}
```
Encoding: Fork and Join

\[
x \mapsto y [a \in x \mapsto a]
\]

\[
x(p,a,l) \; \xrightarrow{\text{def } D} \; D_1 \land D_2
\]

\[
in \; y \cup \{a \in \mathbb{L} \cup \mathbb{O} \cup \mathbb{R} \cup \mathbb{S} \}
\]

\[
| \; z \cup \{a \in \mathbb{L} \cup \mathbb{O} \cup \mathbb{R} \cup \mathbb{S} \}
\]

\[
| \; (a,b,t,a) \cup \{a \in \mathbb{L} \cup \mathbb{O} \cup \mathbb{R} \cup \mathbb{S} \}
\]

\[
| \; \text{state}((a,b,t,a))
\]

\[
x \mapsto y \mapsto z \mapsto x(p,a,l) [y \cup y \cup y \cup y]
\]

\[
\text{def } D
\]

\[
in \; z \cup \{a \in \mathbb{L} \cup \mathbb{O} \cup \mathbb{R} \cup \mathbb{S} \}
\]

\[
| \; (a,b,t,a) \cup \{a \in \mathbb{L} \cup \mathbb{O} \cup \mathbb{R} \cup \mathbb{S} \}
\]

\[
| \; \text{state}((a,b,t,a))
\]

Encoding: Names

\[
\text{def } z(x) = x \text{ in } y \mapsto z
\]

\[
\text{def } z(x) = x \text{ in } y \mapsto z
\]
**Encoding: Names**

\[
\begin{align*}
\text{def } \text{z}(x) &\text{ in } z : 0 \rightarrow [\text{def } z(x) \text{ in } y : 0] \\
\quad &\text{z} \mapsto [y(y) \mid \text{def } z(x) \text{ in } 0 : 0] \\
&\rightarrow y(y)
\end{align*}
\]

\[
\begin{align*}
[\text{def } z(x) \text{ in } z : 0] &\rightarrow [\text{def } z(x) \text{ in } y : 0] \\
\quad &\text{z} \mapsto [y(y) \mid \text{def } z(x) \text{ in } 0 : 0] \\
&\rightarrow y(y)
\end{align*}
\]

**Encoding: Names**

\[
\begin{align*}
[\text{def } z(x) \text{ in } z : 0] &\rightarrow [\text{def } z(x) \text{ in } y : 0] \\
\quad &\text{z} \mapsto [y(y) \mid \text{def } z(x) \text{ in } 0 : 0] \\
&\rightarrow y(y)
\end{align*}
\]

**Encoding: Names**

\[
\begin{align*}
[\text{def } z(x) \text{ in } z : 0] &\rightarrow [\text{def } z(x) \text{ in } y : 0] \\
\quad &\text{z} \mapsto [y(y) \mid \text{def } z(x) \text{ in } 0 : 0] \\
&\rightarrow y(y)
\end{align*}
\]

Definitions have different behaviours depending on received names.
**Encoding: Names**

- A definition is encoded as several definitions:
  
  \[ z(x) \vdash x(x) \]
  
  \[ z'(x, p, a, b) \vdash p \setminus \phi, \{ x(x) \} \]
  
  \[ z''(x, p, a, b) \vdash x(x, p, a, b) \]

- The encoding function takes into account the scope of names: \( [\_ \_ ]_{x,a} \)

---

**Encoding: Merge Names**

- `def (b) \in \{ (b: 0) \}` has two different behaviors:
  - It can commit
  - It can compute with the global reaction rule for \( (b: 0) \)

- Merge definitions are encoded with two rules:
  - One encoding the commit of the thread
  - The other, the application of the rule inside a negotiation

- Moreover, the commit behavior is allowed only when parameters are global names

---

**Correctness and Completeness**

- **Correctness**
  
  \( P: \square_i \) and canonical. If \( P \rightarrow_{\gamma} P' \) with \( P: \square_i \), then \( \exists Q \) s.t. 
  
  \( [P']_{\square P_i \phi} \rightarrow_{\gamma} Q \) and \( \text{norm}(Q) = [P']_{\square P_i \phi} \)

- **Completeness**
  
  \( P: \square_i \) and canonical. If \( [P']_{\square P_i \phi} \rightarrow_{\gamma} Q \) and \( \text{norm}(Q) \) is well-defined, then \( \exists P' \) s.t. \( P \rightarrow_{\gamma} P' \), and \( \text{norm}(Q) = [P']_{\square P_i \phi} \)

---

**Concluding remarks**

- Flat cjoin can be implemented in Join
  - Commit is fully distributed

- This suggests that full cjoin can be modeled back in Join
  - At commit, a sub-negotiation can generate its parent:
    - new threads, and
    - messages to be delivered at commit
  - On abort, sub-negotiations should finish but not compensate

- Extension of running implementations of join (Jocaml, Comega, Join-Java)