CS−Flow: The Engineering of Pervasive Workflows

Hussein Zedan

©Software Technology Research Laboratory (STRL)

Opatija, September 2012
Why?

- A workflow is a set of activities, each performs a piece of functionality within a given context and may be constrained by some security requirements. These activities are coordinated to collectively achieve a required business objective.

- The specification of such coordination is presented as a set of "execution constraints" which include parallelisation, serialisation, restriction, alternation, compensation and so on.
Activities within workflows could be carried out by humans, various software-based application programs, or processing entities according to some organisational rules, such as meeting deadlines or performance improvement.

Workflow execution can involve a large number of different participants, services and devices which may cross the boundaries of various organisations and accessing variety of data.
Modern workflows are CRITICAL systems.

They are

- Highly distributed
- Context-critical
- Security-critical
- Time-critical
- Business-critical

We need a unified model within which modern workflows can be modelled, analysed and, being critical, be provably correct.
Computational Model

our model has three distinct components:

- Context
- Activity and
- Guard
Contexts can take a variety of forms: different platforms and operating systems, hand-held devices, web-services, etc.

A context is characterised by what we call the *context frame*, which is a set of variables (or attributes) of interests.

For

- **PDAs** attributes of interests could be *processor speed, memory size, battery lifetime*.
- a **human** context, *age, qualification, work experience* may be of interest.
- a **patient** context, *body temperature, blood pressure, kidney functions* are more appropriate attributes.
The changes in the attributes are only *observed* and then acted upon.

Context attributes are predicated upon to form a *context guard* so as a decision may be taken to execute an activity or choose different but more suitable context, etc.

Context guards are also important as mechanisms to express *security policies* and for the design of variety of enforcement mechanisms of these policies that, for example, controls access to sensitive data/information.
An activity in our model does not exist in isolation. Indeed it requires a context to house it.

Activities within a workflow move into a context to be executed but may choose to move out to another context in order to complete its functionality.

In this way, context can be nested in a larger context in a compositional fashion.
An activity is a computational unit that describes a piece of work that contributes toward the accomplishment of a given goal.

An activity has

- a goal,
- an input,
- an output,
- performed in a particular order,
- associated with a particular context,
- uses resources/information,
- may affect more than one organisation unit,
- creates some value for users. and
- properly terminates – in the same or in a different context.
An activity starts in one context but may terminate in a different context. This means that an activity has the ability to be **mobile** and moves from one context to another.

But as an activity in our model is tightly associated with a context, mobility occurred at a context level, i.e. **an activity moves with its context**.
Activities may be composed concurrently to produce a new activity which terminates if and only if all of its components terminate, i.e., we adopt the *distributed termination* convention.

- We assume a single clock for an instant of a workflow.
- Activities are also composed in alteration and in a non-deterministic fashions.
- An activity can also be conditionally executed after the passability of its condition or guard.
Guards

- Each activity/context is governed by a set of context and/or security policies/constraints which are continually changing due to either the occurrence of an event and/or the passage of time.

- **access control policies**: **subjects** – such as human, activities, platforms; **object** – This is a resource which is there to be used. It has a state where a subject can alter once it is granted to do so and **action** – is an activity where once the access is granted, it can be executed.

- **ECA** is another formulation of policy.
**CS−F low: Graphical Representation**

\[
\begin{align*}
\text{Context}_id : \langle \text{Frame} \rangle \\
\text{Policies}_\text{Constraints} \\
\| \text{Context}_\text{Constraints} \\
\| \text{Behavioural}_\text{Description}
\end{align*}
\]

where \textit{Frame} is given as:

\[
\text{Frame} :: \langle \text{Context}_\text{Attributes} \rangle
\]
$PDA_1 : \langle s, w, p \rangle$

$(PDA_1(s, w, p) \parallel P_{PDA_1}(Ch, x, y, z))$
CS−Flow: Graphical Representation

Alias: \( (loc, t) \)

\[ PDA_1 : \langle s, w, p \rangle \]

\[ (PDA_1 (s, w, p) \parallel P_{PDA_1} (Ch, x, y, z)) \]

\[ || P_{Ali} \]
\( CS - F low: \) Graphical Representation

\[ PDA_1 : \langle s, w, p \rangle \]

\[ (PDA_1(s, w, p) \parallel P_{PDA_1}(Ch, x, y, z)) \]

\[ PDA_{10} : \langle s_1, w_1, p_1 \rangle \]

\[ P_{PDA_{10}}(Ch, a, b, c) \]
Activities can communicate by exchanging messages over channels.

The communication is synchronous and is modelled using handshake message passing communication primitives: $C!v$ (output) and $C?x$ (input).

$P_{PDA_1} \equiv \ldots ; Ch!Temp_{value} ; \ldots$

and

$P_{PDA_{10}} \equiv \ldots ; Ch?x ; \ldots$
$CS - \textit{Flow}$: Mobility

Warehouse

$PDA_1$

$Q; \text{to (Van)}; R$

$PDA_{10}$

$PDA_3$

Van

$Laptop_2$

$PDA_{31}$
$CS - \mathcal{F}low$: Mobility

Warehouse

$PDA_{10}$ $\parallel$ $PDA_3$

Van

$PDA_1$ $\parallel$ $Laptop_2$ $\parallel$ $PDA_{31}$
**CS–Flow: textual Representation**

\[
P, Q ::= \text{skip} | \text{abort} | x := v | \text{delay}(t) | [t_1 \ldots t_n] P | c ! v | c ? x
\]

\[
| \alpha \langle \vec{x} \rangle : \{P\} | \text{to}(\alpha) | \text{var} \vec{x} \text{ in } P \{Q\} | \text{chan} \vec{c} \text{ in } P \{Q\}
\]

\[
| \text{in} \alpha \cdot P(\vec{x}) | P ; Q | P \parallel Q | P \triangleright^G Q | \text{while } G \cdot \text{ do } P \text{ od}
\]

\[
| [p_1] : G_1 \rightarrow P \square [p_2] : G_2 \rightarrow Q
\]

\[
G ::= \text{true} | b | \text{not } G | G_1 \text{ and } G_2 | \text{somewhere}(\alpha) \cdot G
\]
\[ \alpha \langle \tilde{x} \rangle : \]
\[
\{
\}
\]
\[ P \]
\[
\}
\]
\textit{Ikea}:

\[ \langle \rangle \]

\[ \{ \]

\[ P_{Ikea} \parallel PDA_{23} : \{ Q \} \]

\[ \} \]
Ikea : \langle \text{Damp}_{\text{Level}}, \text{Smoke}_{\text{Alarm}} \rangle

\{ 
\quad P_{\text{Ikea}} \parallel P_{\text{DA23}} : \{ Q \} 
\}

CS-Flow: The Engineering of Pervasive Workflows
Ikea : \langle Damp_{Level}, Smoke_{Alarm} \rangle

\{ 
   P_{Ikea} \parallel PDA_{23} : 
   \{ 
      TakeStock ; 
      \text{not} \ (Damp_{Level} \geq 25 \lor Smoke_{Alarm}) \rightarrow \text{to}(Van) ; Place Order 
   \} 
\}
Central to our model is that activities do not operate in the ether. They need contexts which identify their *locations* and within which they execute, terminate and may move out of them to another contexts.

Unlike other formalisms, the notion of *holes* exists in which processes can move to. This makes the models rather clumsy and static with a fixed number of holes.
The term "context" is used here instead of "location" for the later can indicate/require notions such as

- Proximity,
- Coordinates,
- Neighborhoods, etc.

which in our view adds extra complication which is not needed.
Two special contexts, which we call **SKIP** and **STOP**:

- **SKIP** is an empty context and nothing is happening in it and there are no observables.
- **STOP** is the most un-inhabited context and will remain so forever! Further, if it moves into another context, it makes the host context un-inhabitable too. It is a context that needs to be avoided at all cost.
Context, like activities, can communicate synchronously via channels. Whenever a context moves, its channels move with it. This is a powerful mobility notion as all what we needed is a single label to identify a context. The connectivity’s between contexts (or their exact coordinates, neighborhoods, etc.) becomes irrelevant.
Examples: Adaptable activities

ShopFloor:

\{ 

\text{win} \parallel \text{linx}

\}

Hussein Zedan

\text{CS-Flow: The Engineering of Pervasive Workflows}
Examples: Adaptable activities

\[
\text{\textit{win}}:
\
\{ \\
\quad \text{\textit{var} } f \text{ in } \text{\textit{edit}} \\
\quad \{ \\
\quad\quad \text{\textit{notepad}}(f) \\
\quad\}
\}
\]

Hussein Zedan

CS-Flow: The Engineering of Pervasive Workflows
Examples: Adaptable activities

\[\text{linx} : \]
\[
\{ \\
\text{var } f \text{ in } \text{edit} \\
\{ \\
\text{emacs}(f) \\
\}
\}
\]
Examples: Adaptable activities

Employee:

\{
   \text{somewhere} (\text{ShopFloor}) \cdot \text{edit}(\text{file})
\}
Examples: Adaptable activities

\[
\text{win} : \\
\{ \\
\text{var} \ f \ \text{in} \ \text{edit} \\
\{ \\
\ \text{notepad}(f) \\
\} \\
\} \\
\| \\
\text{Employee} : \\
\{ \\
\ \text{somewhere}(\text{ShopFloor}) \cdot \text{edit}(\text{file}) \\
\} \\
\}
Examples: Adaptable activities

\[ \text{linx : } \]
\[
\{ \\
\quad \text{var } f \text{ in } \text{edit} \\
\quad \{ \\
\quad\quad \text{emacs}(f) \\
\quad\} \\
\} \\
\| \]

\[ \text{Employee : } \]
\[
\{ \\
\quad \text{somewhere}(\text{ShopFloor}) \cdot \text{edit(file)} \\
\} \\
\} \]
while true

do
{

\[ G_{event_1} \text{ and } G_{condition_1} \rightarrow P \]

\[ \square \]

\[ G_{event_2} \text{ and } G_{condition_2} \rightarrow Q \]

\[ \ldots \]

\[ \square \]

} od
Examples: Policies – ECA

System $\Rightarrow$

$Flows \parallel EventAnalyser \parallel ECA$
Without lose of generality, we assume that

- There is only one parallel operator, $\parallel$, in our system. Nesting concurrency can be dealt with by applying the transformation to the most inner $\parallel$ and continue to move to the outer constructs.
- The length of all activities in the system are the same. This can be easily achieved using the semantics of $\text{skip}$. I.e.

$$\text{skip} ; S \equiv S ; \text{skip} \equiv S$$
\( C_1 \langle \tilde{a} \rangle : \)

\[
\begin{align*}
\{ & \\
\text{var } \tilde{x}, \tilde{y} \text{ in} & \\
\{ & \\
 P_1; & \\
 P_2; & \\
 P_3; & ||
\end{align*}
\]

\( C_2 \langle \tilde{b} \rangle : \)

\[
\begin{align*}
\{ & \\
\text{var } \tilde{x}_1, \tilde{y}_1 \text{ in} & \\
\{ & \\
 Q_1; & \\
 Q_2; & \\
 Q_3; &
\end{align*}
\]

\( \)
\[ C_1 \langle \tilde{a} \rangle : \]
\[
\{ \]
\[
\text{var} \, \tilde{x}, \tilde{y} \text{ in} \]
\[
\{ \]
\[
P_1; \]
\[
P_2; \]
\[
P_3; \parallel \]
\[
P_4; \]
\[
P_5; \]
\[
\} \]
\[
\} \]
\[
\}
\]
\[ C_2 \langle \tilde{b} \rangle : \]
\[
\{ \]
\[
\text{var} \, \tilde{x}_1, \tilde{y}_1 \text{ in} \]
\[
\{ \]
\[
\text{skip;} \]
\[
\text{Q}_1; \]
\[
\text{skip;} \]
\[
\text{Q}_2; \]
\[
\text{Q}_3; \]
\[
\} \]
\[
\} \]
\[
\} \]
Let us consider an example:

\[ \mathcal{R} \equiv C_1 \langle \tilde{a} \rangle : \{ \mathcal{P} \} \parallel C_1 \langle \tilde{b} \rangle : \{ \mathcal{Q} \} \]

\[
\begin{align*}
C_1 \langle \tilde{a} \rangle : & \\
\{ & \\
\text{var } x, y, z, \text{ chan}_1 \text{ in} & \\
\{ & \\
y := y + x; & \\
z := y \times z; & \\
\text{chan}_1 ! z; & \\
\} & \\
\} & \\
\end{align*}
\]

\[
\begin{align*}
C_2 \langle \tilde{b} \rangle : & \\
\{ & \\
\text{var } x_1, \text{ chan}_1 \text{ in} & \\
\{ & \\
\text{chan}_1 ? x_1; & \\
x_1 := x_1 \times x_1; & \\
\} & \\
\} & \\
\end{align*}
\]
**Definition**

A layer, $L$, of a workflow, $S$, is a logical horizontal partition that cuts across all concurrent threads of $S$.

**Definition**

A Layer $L$ is called communicating layer if it contains at least one communication primitive. It is called communication-closed if a communication starts and terminates in the same layer.

A non-communicating layer is that which contains no communication primitives.

**Definition**

A super-structure over a workflow, $S$, is a quasi-sequential composition of layers from $S$. 
\[
L_1 \equiv y := y + x \quad \parallel \quad chan_1 ? x_1
\]

\[
L_2 \equiv z := y \times z \quad \parallel \quad x_1 := x_1 \times x_1
\]

\[
L_3 \equiv chan_1 ! z \quad \parallel \quad skip
\]

\[
L_4 \equiv
\begin{align*}
y &:= y + x; \quad \text{skip;}
\end{align*}
\]

\[z := y \times z \quad \text{skip}
\]

\[
L_5 \equiv
\begin{align*}
z &:= y \times z; \quad \text{skip;}
chan_1 ! z; \quad \parallel \quad chan_1 ? x_1;
skip \quad \quad x_1 := x_1 \times x_1
\end{align*}
\]
The following are some super-structures:

1. $S_{R_1} \equiv R$
2. $S_{R_2} \equiv L_1 ; L_2 ; L_3$
3. $S_{R_3} \equiv L_4 ; L_5$

Under what condition(s) will a super-structure workflow be equivalent to the original one?
It is clear that, in the example above, $L_2$ and $L_4$ are non-communicating layers while $L_1$, $L_3$ and $L_5$ are communication-closed. $S_{R_3}$ and $S_{R_4}$ are a quasi-sequential workflow whilst $S_{R_2}$ is not.
Theorem

For any $CS-\mathcal{F}$ low workflow system $S$ there exist a semantically equivalent quasi-sequential system, $S_L$. 
Proof: Choices

\[ G_1 \rightarrow P \]

\[ \square \]

\[ G_2 \rightarrow Q \]
Proof: Choices

The following workflow, \( S' \), is a such safe decomposition:

\[
S' \equiv \begin{cases}
G_1 \rightarrow P; \text{GFlag} := \text{false} \\
\quad \square \\
G_2 \rightarrow \text{GFlag} := \text{true}
\end{cases}; \quad (S_{11})
\]

\[
\begin{cases}
\text{GFlag} \rightarrow Q \\
\quad \square
\end{cases} \quad (S_{12})
\]

Now, if we have another workflow \( D \) of the same structure as \( S \) then

\[
S \parallel D
\]

can be "safely" decomposed into the structure:

\[
((S_{11} \parallel D_{11}) ; (S_{12} \parallel D_{12})) \square ((S_{21} \parallel D_{21}) ; (S_{22} \parallel D_{22}))
\]

where each \( S_{ij} \), for all \( i,j = 1,2 \), is either a non-communicating layer or a communication-closed layer.
Proof: Choices

This can be rewritten as

\[
\left((S_{11} \parallel D_{11}) \Box (S_{21} \parallel D_{21})\right);
\]
\[
\left((S_{11} \parallel D_{11}) \Box (S_{22} \parallel D_{22})\right);
\]
\[
\left((S_{12} \parallel D_{12}) \Box (S_{21} \parallel D_{21})\right);
\]
\[
\left((S_{12} \parallel D_{12}) \Box (S_{22} \parallel D_{22})\right)
\]

These structures demonstrate that layers can be composed, respectively, as a series of alternative or sequentially. In fact, structures such as iteration, conditional, interrupt, etc. can also be used.
Proof: Iterations

We assume that

1. Loops are *finite*

2. Communication symmetry is assured (i.e., communication-deadlock free)

It should be noted that due to (1) above, a finite loop can be replaced as a set of sequentially composed statements and because of (2), we can always ensure (using `skip`) that each layer is communication-closed layer.
Proof: Iterations

Using

\[ \text{while } G \text{ do}\{P\} \equiv G \rightarrow P ; (\text{while } G \text{ do}\{P\}) \]

then, if we have

\[ \text{while } G \text{ do}\{P\} \parallel Q \]

Then we can transform this to the semantically equivalent \(CS-Flow\) system

\[ (((G \rightarrow P) \parallel Q) ; (\text{while } G \text{ do}\{P\}) \]

Then, we layer \(((G \rightarrow P) \parallel Q)\) into communication-closed layers (depending on the structure of \(P\) and \(Q\), and repeat the process on the \(\text{while } G \text{ do}\{P\}\), and so on.
Layers: Fault-Tolerance

The diagram illustrates the layers of fault-tolerance in a system. Each layer, denoted as $L_i$, represents a stage in the workflow, with $P$, $Q$, and $E$ being transition points. The processes include Save, Commit, Rollback, and Backward recovery. The diagram shows the flow from $P$ to $Q$, with potential rollback points and recovery mechanisms identified at $E$.
Layers: Fault-Tolerance

- Save
- Commit
- Forward recovery
- Rollback to the top
- Error exception

Hussein Zedan

*CS-Flow*: The Engineering of Pervasive Workflows
The idea here is to transform the existing $CS-Flow$ design into a semantically equivalent communication-closed layer design in which the analyses are easier than the original one. The rational is that the resulting layer-design is quasi-sequential and hence all existing formalisms for sequential systems can be deployed.
Layers Design Methodology

1. **Requirements Decomposition.**
   - Decompose the given workflow requirements into a number of sub-requirements which can be as fine or coarse grain as we wish. This process *iterative* in nature.
   - Experience has shown that, identifying, what we call *Actors* helps in specifying layer interfaces.

2. **Layer Design.**
   - Design layers which conform/satisfy its requirement.
   - The layers however have to be *communication-closed* layers.

3. **Integration.**
   - Compose/integrate all layers into a complete $CS-Flow$ workflow.
Example: One-place Buffer – Layer Decomposition

We can easily identify two major layers:

1. *Initialisation*. Involves the *Buffer* and the *Authorised User*, and using channel *push*.

2. *Operations*. This involves the *Buffer* and any other user.
Init $\triangleq$

\[
push!v \parallel (push?x;\: empty:=\:false) \parallel \: \text{skip}
\]
Example: One-place Buffer – Layer Design

\[
\text{Operation} \triangleq ( \\
\text{skip} \parallel \\
\text{while true do} \\
\{ \\
\quad \text{empty} \rightarrow \text{push} ? x \\
\quad \Box \\
\quad \text{not empty} \rightarrow \text{pull} ! v \\
\} \\
\text{od} \\
\parallel \\
\text{pull} ? x ; \\
\text{push} ! v)
\]
Example: One-place Buffer – Layer Design

It is clear that each of the above layers are communication-closed and the resulting quasi-sequential system is

$$Sys_L \triangleq Init \; ; \; Operation$$
Example: One-place Buffer – Integration

In this phase, the layers are integrated to obtain the final system:

\[ Sys \equiv Authorised\_User \parallel Buffer \parallel User \]

where

\[ Buffer \equiv ( \]

\[ \text{push} \ ? \ x ; \]
\[ \text{empty} := \text{false} ; \]

while true

\[ \text{do} \]

\[ \{ \]

\[ \text{empty} \to \text{push} \ ? \ x \]

\[ \]

\[ \text{not empty} \to \text{pull} \ ! \ v ; \]

\[ \text{empty} := \text{true} \]

\[ \} \]

Hussein Zedan

\textit{CSFlow: The Engineering of Pervasive Workflows}
The users are modelled as

\[ Users :: ( \]

\[ \text{Authorised}\_\text{User} \triangleq \text{push} ! v \]

\[ \text{User} \triangleq \text{pull} ? x ; \text{push} ! v \]

We note that, as the layers were designed communication-closed, then \( Sys\_L \equiv Sys \).
Context-Aware Ward: CAW

Hussein Zedan

CS−Flow: The Engineering of Pervasive Workflows
\[ \text{CAW} \triangleq ( \]

\[ \text{nurse} \langle \tilde{x}_n \rangle : \{P_n\} \]

\[ \parallel \text{bed} \langle \tilde{w} \rangle : \{P_b\} \]

\[ \parallel \text{patient} \langle \tilde{w}_1 \rangle : \{P_p\} \]

\[ \parallel \text{nurse - office} \langle \tilde{w}_2 \rangle : \{P_{n.o}\} \]

\[ \parallel \text{medicine - room} \langle \tilde{w}_3 \rangle : \{P_{m.r}\} \]

\[ \parallel \]

\[ \text{tray} \langle \tilde{x}_t \rangle : \{P_t \parallel \text{Cont}_1 \langle \text{pat}_1, \tilde{a}_1 \rangle : \{P_1\} \]

\[ \left( \begin{array}{c}
\parallel \text{Cont}_2 \langle \text{pat}_2, \tilde{a}_2 \rangle : \{P_2\} \\
\parallel \ldots \ldots \\
\parallel \text{Cont}_k \langle \text{pat}_k, \tilde{a}_k \rangle : \{P_k\} \\
\end{array} \right) \]

\[ ) \]
\[ P_n \overset{\cong}{=} \text{chan } \text{chan}_{n.t} \text{ in} \{
\]

while true

\[ \text{chan}_{n.t} \text{ ! any ;} \]

\[ \text{to}(\text{bed}) ; \]

\[ [\text{epr}(P_i) \parallel \text{HandOutDrug}(P_i)] \]

\[ \text{chan}_{n.t} \text{ ! any ;} \]

\[ \text{to}(\text{nurse} - \text{office}) \]
HandOutDrug \equiv \text{var } T, D, N, i \text{ in } \{

\text{while } i \leq N 

\text{do }

\quad ([T][D]\text{GiveDrug}) \parallel \text{delay}(T)

\quad i = i + 1

\text{od}

\}
In addition to equational theory and operational semantics for $CS-Flow$ we have

- Denotational semantics: A CCA-specification semantics
- Reduction semantics
Reduction Rules

\[
P \rightarrow P' \Rightarrow \text{var } \tilde{x} \text{ in } \{P\} \rightarrow \text{var } \tilde{x} \text{ in } \{P'\} \quad \text{(Reduction Var)}
\]

\[
P \rightarrow P' \Rightarrow \text{chan } \tilde{x} \text{ in } \{P\} \rightarrow \text{chan } \tilde{x} \text{ in } \{P'\} \quad \text{(Reduction Chan)}
\]

\[
P \rightarrow P' \Rightarrow \alpha < \tilde{x} >: \{P\} \rightarrow \alpha < \tilde{x} >: \{P'\} \quad \text{(Reduction Contxt)}
\]

\[
P \rightarrow P' \Rightarrow C(P) \rightarrow C(P') \quad \text{(Reduction Context)}
\]

\[
P \rightarrow P' \Rightarrow P \parallel Q \rightarrow P' \parallel Q \quad \text{(Reduction Par)}
\]

\[
P \equiv Q, Q \rightarrow Q', Q' \equiv P' \Rightarrow P \rightarrow P' \quad \text{(Reduction } \equiv \text{)}
\]
Reduction Rules

\[(Chan \ ? \ \tilde{y}) \ ; \ P \parallel (Chan \ ! \ \tilde{z}) \ ; \ Q\]

\[\rightarrow \ P\{\tilde{y} \leftarrow \tilde{z}\} \parallel Q\]

(Reduction Com-1)

\[\alpha : \{(Chan \ ? \ \tilde{y}) \ ; \ P\parallel Q\} \parallel \beta : \{(Chan \ ! \ \tilde{z}) \ ; \ R\parallel S\}\]

\[\rightarrow \ \alpha : \{P(\tilde{y} \leftarrow \tilde{z}) \parallel Q\} \parallel \beta : \{R \parallel S\}\]

(Reduction Com-2)

\[\alpha : ((Chan \ ? \ \tilde{y}) \ ; \ P\parallel Q) \parallel \beta : (\alpha : (Chan \ ! \ \tilde{z}) \ ; \ R\parallel S)\]

\[\rightarrow \ \alpha : (P(\tilde{y} \leftarrow \tilde{z})) \parallel \beta : (R \parallel S)\]

(Reduction Com-3)

\[\alpha : (\beta : (Chan \ ? \ \tilde{y}) \ ; \ P\parallel Q) \parallel \beta : (Chan \ ! \ \tilde{z} \ ; \ R\parallel S)\]

\[\rightarrow \ \alpha : (P(\tilde{y} \leftarrow \tilde{z}) \parallel Q) \parallel \beta(R \parallel S)\]

(Reduction Com-4)

\[\alpha : (\beta : (Chan \ ? \ \tilde{y}) \ ; \ P\parallel Q) \parallel \beta : (\alpha : ((Chan \ ! \ \tilde{z}) \ ; \ R\parallel S))\]

\[\rightarrow \ \alpha : (P(\tilde{y} \leftarrow \tilde{z}) \parallel Q) \parallel \beta(R \parallel S)\]

(Reduction Com-5)

\[\beta : \{to(\alpha) \ : \ P\parallel Q\} \parallel (\alpha : \{R\})\]

\[\rightarrow \ \alpha : \{\beta : \{P \parallel Q\} \ ; \ R\}\]

(Reduction Mob)
buf[
\@recv(x).send(x).0 |
\@recv(y).send(y).@recv(z).send(z).0
]