CoMingle: Distributed Logic Programming for Decentralized Mobile Ensemble

Edmund S. L. Lam   Iliano Cervesato
sllam@qatar.cmu.edu   iliano@cmu.edu

Carnegie Mellon University

Supported by QNRF grant JSREP 4-003-2-001

March 2015

Outline

Distributed Programming

- *Computations that run at more than one place at once*
  - A 40 year old paradigm
  - Now more popular than ever
    - Cloud computing
    - Modern webapps
    - ***Mobile device applications***
- Hard to get right
  - Concurrency bugs (race conditions, deadlocks, . . .)
  - Communication bugs
  - "Normal" bugs
- Two views
  - *Node-centric* — program each node separately
  - *System-centric* — program the distributed system as a whole
    - Compiled to node-centric code
    - Used in limited settings (Google Web Toolkit, MapReduce)

What is CoMingle?

A programming language for distributed mobile apps

- Declarative, concise, based on linear logic
- Enables high-level *system-centric* abstraction
  - **specifies** distributed computations as *ONE* declarative program
  - **compiles** into node-centric fragments, executed by each node
- Designed to implement mobile apps that run across Android devices
- Inspired by CHR [Frühwirth and Raiser, 2011], extended with
  - Decentralization [Lam and Cervesato, 2013]
  - Comprehension patterns [Lam and Cervesato, 2014]
- Also inspired by Linear Meld [Cruz et al., 2014]

Outline

CoMingle by Example

```
module comingle.lib.ExtLib import {
     size   :: A -> int.
}

predicate swap    :: (loc,int) -> trigger.
predicate item    :: int -> fact.
predicate display :: (string,A) -> actuator.

rule pivotSwap :: [X]swap(Y,P),
                  {[X]item(D)|D->Xs. D >= P},
                  {[Y]item(D)|D->Ys. D <= P}
                    --o [X]display(Msg,size(Ys),Y), {[X]item(D)|D<-Ys},
                        [Y]display(Msg,size(Xs),X), {[Y]item(D)|D<-Xs}
                        where Msg = "Received %s items from %s".
```

CoMingle by Example: Decentralized Multiset Rewriting

```
[X]swap(Y,P)
{[X]item(D)|D->Xs.D>=P} --o [X]display(Msg,size(Ys),Y), {[X]item(D)|D<-Ys}
{[Y]item(D)|D->Ys.D<=P}     [Y]display(Msg,size(Xs),X), {[Y]item(D)|D<-Xs}
                            where Msg = "Received %s items from %s".
```

Let s = swap, i = item and d = display

Node: $n1$		Node: $n2$	Node: $n3$
$s(n2, 5), i(4), i(6), i(8)$	$\longrightarrow$	$i(3), i(20)$	$s(n2, 10), i(18)$

CoMingle by Example: Decentralized Multiset Rewriting

```
[X]swap(Y,P)
{[X]item(D)|D->Xs.D>=P} --o [X]display(Msg,size(Ys),Y), {[X]item(D)|D<-Ys}
{[Y]item(D)|D->Ys.D<=P}     [Y]display(Msg,size(Xs),X), {[Y]item(D)|D<-Xs}
                            where Msg = "Received %s items from %s".
```

Let s = swap, i = item and d = display

Node: $n1$
$s(n2, 5), i(4), i(6), i(8)$

$\longrightarrow$

Node: $n2$
$i(3), i(20)$

Node: $n3$
$s(n2, 10), i(18)$

$\downarrow$

Node: $n1$
$d(\text{"1 from n2"})$
$i(3), i(4)$

Node: $n2$
$d(\text{"2 from n1"})$
$i(6), i(8), i(20)$

Node: $n3$
$s(n2, 10), i(18)$

CoMingle by Example: Decentralized Multiset Rewriting

```
[X]swap(Y,P)
{[X]item(D)|D->Xs.D>=P} --o [X]display(Msg,size(Ys),Y), {[X]item(D)|D<-Ys}
{[Y]item(D)|D->Ys.D<=P}     [Y]display(Msg,size(Xs),X), {[Y]item(D)|D<-Xs}
                            where Msg = "Received %s items from %s".
```

Let s = swap, i = item and d = display

Node: $n1$

$s(n2, 5), i(4), i(6), i(8)$

→

Node: $n2$

$i(3), i(20)$

Node: $n3$

$s(n2, 10), i(18)$

↧

Node: $n1$

$d(\text{"1 } from\ n2\text{"})$
$i(3), i(4)$

Node: $n2$

$d(\text{"2 } from\ n1\text{"})$
$i(6), i(8), i(20)$

←

Node: $n3$

$s(n2, 10), i(18)$

CoMingle by Example: Decentralized Multiset Rewriting

```
[X]swap(Y,P)
{[X]item(D)|D->Xs.D>=P} --o [X]display(Msg,size(Ys),Y), {[X]item(D)|D<-Ys}
{[Y]item(D)|D->Ys.D<=P}     [Y]display(Msg,size(Xs),X), {[Y]item(D)|D<-Xs}
                            where Msg = "Received %s items from %s".
```

Let s = swap, i = item and d = display

Node: $n1$
$s(n2, 5), i(4), i(6), i(8)$

Node: $n2$
$i(3), i(20)$

Node: $n3$
$s(n2, 10), i(18)$

Node: $n1$
$d(\text{"1 } \textit{from n2}\text{"})$
$i(3), i(4)$

Node: $n2$
$d(\text{"2 } \textit{from n1}\text{"})$
$i(6), i(8), i(20)$

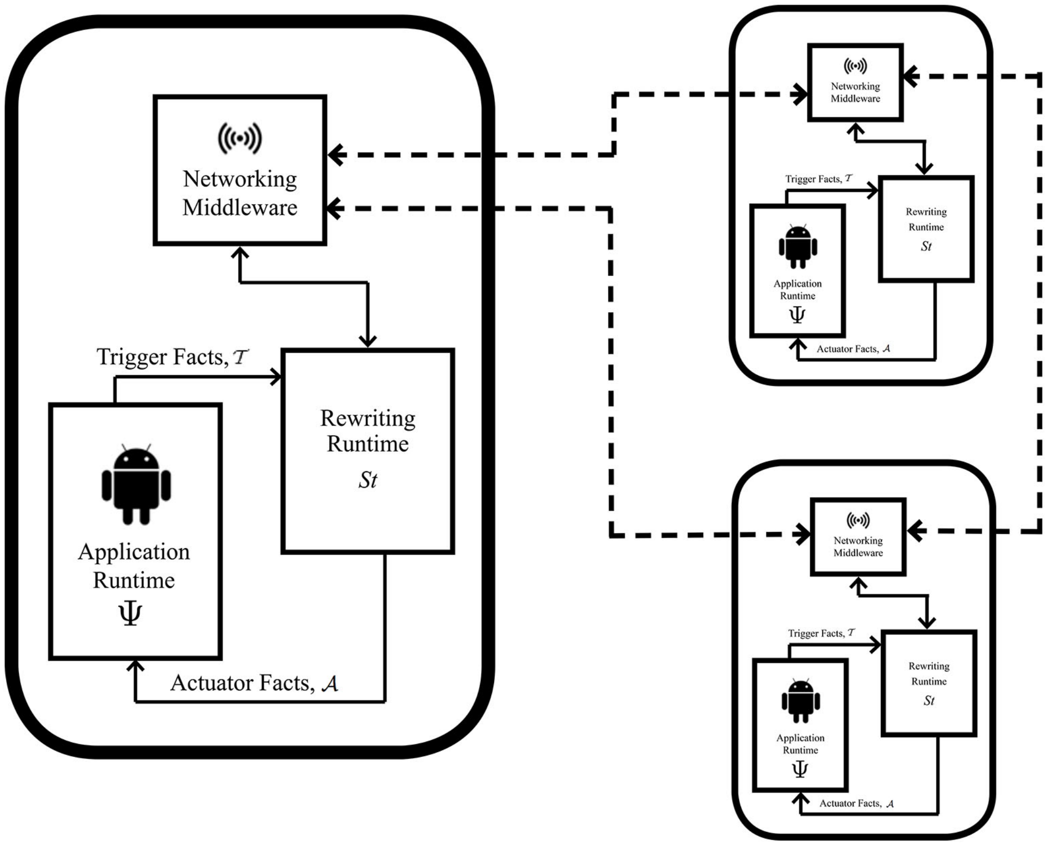
Node: $n3$
$s(n2, 10), i(18)$

Node: $n1$
$d(\text{"1 } \textit{from n2}\text{"})$
$i(4), i(3)$

Node: $n2$
$d(\text{"2 } \textit{from n1}\text{"})$
$d(\text{"1 } \textit{from n3}\text{"})$
$i(18), i(20)$

Node: $n3$
$d(\text{"2 } \textit{from n2}\text{"})$
$i(6), i(8)$

CoMingle Architecture



CoMingle by Example: Triggers and Actuators

```
predicate swap    :: (loc,int) -> trigger.
predicate item    :: int -> fact.
predicate display :: (string,A) -> actuator.

rule pivotSwap :: [X]swap(Y,P),
                  {[X]item(D)|D->Xs. D >= P},
                  {[Y]item(D)|D->Ys. D <= P}
                    --o [X]display(Msg,size(Ys),Y), {[X]item(D)|D<-Ys},
                        [Y]display(Msg,size(Xs),X), {[Y]item(D)|D<-Xs}
                        where Msg = "Received %s items from %s".
```

- **Abstracts** communications between node (i.e., X, Y)
- Executed by a **rewriting runtime** on each node
- Interacts with a local **application runtime** on each node
- Triggers: **inputs** from the application runtime
- Actuators: **outputs** into the application runtime

CoMingle by Example: Triggers and Actuators

```
predicate swap    :: (loc,int) -> trigger.
predicate item    :: int -> fact.
predicate display :: (string,A) -> actuator.

rule pivotSwap :: [X]swap(Y,P),
                  {[X]item(D)|D->Xs. D >= P},
                  {[Y]item(D)|D->Ys. D <= P}
                    --o [X]display(Msg,size(Ys),Y), {[X]item(D)|D<-Ys},
                        [Y]display(Msg,size(Xs),X), {[Y]item(D)|D<-Xs}
                        where Msg = "Received %s items from %s".
```

- Predicate swap is a trigger
  - An input interface into the rewriting runtime
  - Only in rule heads
  - swap(Y,P) is added to rewriting state when button on device X is pressed

CoMingle by Example: Triggers and Actuators

```
predicate swap    :: (loc,int) -> trigger.
predicate item    :: int -> fact.
predicate display :: (string,A) -> actuator.

rule pivotSwap :: [X]swap(Y,P),
                  {[X]item(D)|D->Xs. D >= P},
                  {[Y]item(D)|D->Ys. D <= P}
                    --o [X]display(Msg,size(Ys),Y), {[X]item(D)|D<-Ys},
                        [Y]display(Msg,size(Xs),X), {[Y]item(D)|D<-Xs}
                        where Msg = "Received %s items from %s".
```

- Predicate `display` is an `actuator`
  - An output interface from the rewriting runtime
  - Only in rule body
  - `display("2 from n1")` executes a screen display callback function

CoMingle by Example: Triggers and Actuators

```
predicate swap    :: (loc,int) -> trigger.
predicate item    :: int -> fact.
predicate display :: (string,A) -> actuator.

rule pivotSwap :: [X]swap(Y,P),
                  {[X]item(D)|D->Xs. D >= P},
                  {[Y]item(D)|D->Ys. D <= P}
                    --o [X]display(Msg,size(Ys),Y), {[X]item(D)|D<-Ys},
                        [Y]display(Msg,size(Xs),X), {[Y]item(D)|D<-Xs}
                        where Msg = "Received %s items from %s".
```

- Predicate `item` is a standard `fact`
  - Can appear in rule head or body
  - Atoms of the rewriting state

CoMingle by Example

```
[X]swap(Y,P)
{[X]item(D)|D->Xs.D>=P} --o [X]display(Msg,size(Ys),Y), {[X]item(D)|D<-Ys}
{[Y]item(D)|D->Ys.D<=P}     [Y]display(Msg,size(Xs),X), {[Y]item(D)|D<-Xs}
                            where Msg = "Received %s items from %s".
```

- High-level specification of distributed triggers/actuators

Distributed Triggers + *Distributed* State Patterns	--o	*Distributed* Actuators + *Distributed* State Patterns

- Declarative, concise and executable!
- Abstracts away
  - Low-level message passing
  - Synchronization
- Ensures atomicity and isolation

Outline

Abstract Syntax

- A CoMingle program $\mathcal{P}$ is a set of rules of the form

$$r : \ H_p \setminus H_s \mid g \ \multimap \ B$$

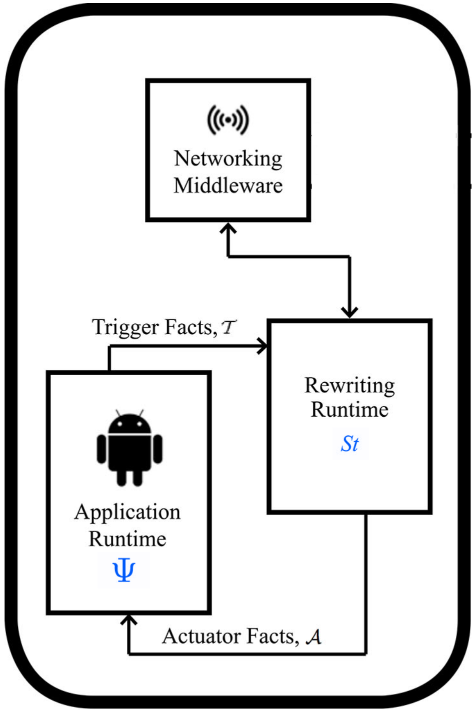
  - H_p, H_s and B: Multisets of patterns
  - g: Guard conditions

- A pattern is either
  - a fact: $[\ell]p(\vec{t})$
  - a comprehension: $\wr[\ell]p(\vec{t}) \mid g\int_{\vec{x}\in t}$

- Three kinds of facts
  - Triggers (only in H_p or H_s): Inputs from the "Android world"
  - Actuators (only in B): Outputs to the "Android world"
  - Standard facts: Atoms of rewriting state

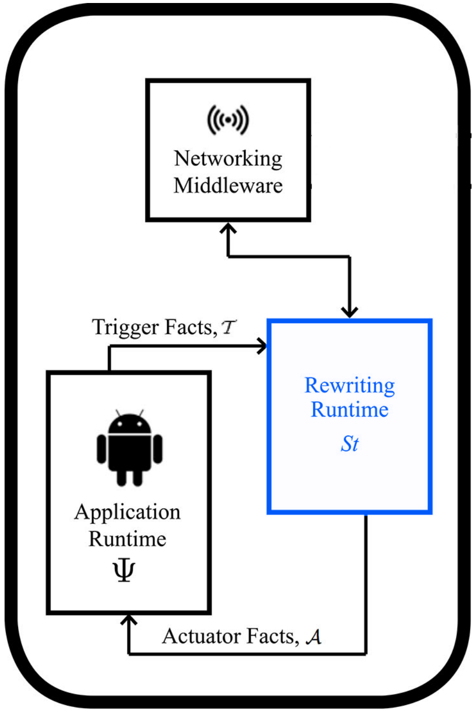
Semantics of CoMingle: Abstract State Transitions

- *CoMingle state* $\langle St; \Psi \rangle$ represents the mobile ensemble
  - St is the *rewriting state*, a multiset of ground facts $[\ell]f$
  - Ψ is the *application state*, a set of local states $[\ell]\psi$
  - A location ℓ is a computing node



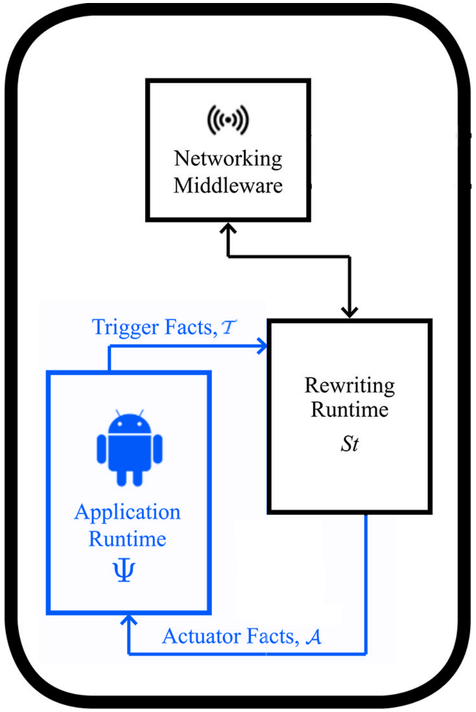
Semantics of CoMingle: Abstract State Transitions

- *CoMingle state* $\langle St; \Psi \rangle$ represents the mobile ensemble
    - St is the *rewriting state*, a multiset of ground facts $[\ell]f$
    - Ψ is the *application state*, a set of local states $[\ell]\psi$
    - A location ℓ is a computing node
- The rewrite runtime: $\mathcal{P} \rhd \langle St; \Psi \rangle \mapsto \langle St'; \Psi \rangle$
    - Applies a rule in $\mathcal{P}$
    - Several locations may participate
    - Decentralized multiset rewriting



Semantics of CoMingle: Abstract State Transitions

- *CoMingle state* $\langle St; \Psi \rangle$ represents the mobile ensemble
  - St is the *rewriting state*, a multiset of ground facts $[\ell]f$
  - Ψ is the *application state*, a set of local states $[\ell]\psi$
  - A location ℓ is a computing node
- The rewrite runtime: $\mathcal{P} \rhd \langle St; \Psi \rangle \mapsto \langle St'; \Psi \rangle$
  - Applies a rule in $\mathcal{P}$
  - Several locations may participate
  - Decentralized multiset rewriting
- The application runtime: $\langle \mathcal{A}; \psi \rangle \mapsto_{\ell} \langle \mathcal{T}; \psi' \rangle$
  - Models local computation within a node
  - All within location ℓ



Rewriting Runtime: Overview

- Decentralized semantics [Lam and Cervesato, 2013]
  - Facts are explicitly annotated with locations, $[\ell]p(\vec{t})$
  - System-centric decentralized multiset rewriting
  - Compiled into node-centric specifications

Rewriting Runtime: Overview

- Decentralized semantics [Lam and Cervesato, 2013]
  - Facts are explicitly annotated with locations, $[\ell]p(\vec{t})$
  - System-centric decentralized multiset rewriting
  - Compiled into node-centric specifications
- Comprehension patterns [Lam and Cervesato, 2014]

$$\wr [\ell]p(\vec{t}) \mid g \int_{\vec{x} \in T}$$

  - Multiset of *all* $[\ell]p(\vec{t})$ in the state that satisfy g
  - $\vec{x}$ bound in g and $\vec{t}$
  - T is the multiset of all bindings $\vec{x}$
  - Semantics enforces maximality of T

Comprehension Example: Pivoted Swapping

$$
\begin{array}{lcl}
 & & [X]\underline{swap}(Y, P) \\
pivotSwap & : & \wr[X]item(D) \mid D \geq P\int_{D \in Xs} \quad \multimap \quad \wr[Y]item(D)\int_{D \in Xs} \\
 & & \wr[Y]item(D) \mid D \leq P\int_{D \in Ys} \qquad\quad\;\; \wr[X]item(D)\int_{D \in Ys}
\end{array}
$$

- Xs and Ys built from the rewriting state — *output*
- Xs and Ys used to unfold the comprehensions — *input*

- Atomic

Rewriting Runtime: Semantics of Matching

- Matching Judgment: $\overline{H} \triangleq_{\textbf{lhs}} St$
  - Matches rule left-hand side $\overline{H}$ against rewriting state St

$$\frac{\overline{H} \triangleq_{\textbf{lhs}} St \quad H \triangleq_{\textbf{lhs}} St'}{\overline{H}, H \triangleq_{\textbf{lhs}} St, St'} \qquad \frac{}{\varnothing \triangleq_{\textbf{lhs}} \varnothing} \qquad \frac{}{F \triangleq_{\textbf{lhs}} F}$$

$$\frac{[\vec{t}/\vec{x}]f \triangleq_{\textbf{lhs}} F \quad \models [\vec{t}/\vec{x}]g \quad \wr f \mid g \int_{\vec{x} \in \overline{ts}} \triangleq_{\textbf{lhs}} St}{\wr f \mid g \int_{\vec{x} \in \vec{t}, \overline{ts}} \triangleq_{\textbf{lhs}} St, F} \qquad \frac{}{\wr f \mid g \int_{\vec{x} \in \varnothing} \triangleq_{\textbf{lhs}} \varnothing}$$

Rewriting Runtime: Semantics of Matching

- Residual Non-matching: $\overline{H} \triangleq^{\neg}_{\mathbf{lhs}} St$
  - Checks that $\overline{H}$ matches nothing (else) in St
  - Ensures maximality

$$\frac{\overline{H} \triangleq^{\neg}_{\mathbf{lhs}} St \quad H \triangleq^{\neg}_{\mathbf{lhs}} St}{\overline{H}, H \triangleq^{\neg}_{\mathbf{lhs}} St} \qquad \frac{}{\emptyset \triangleq^{\neg}_{\mathbf{lhs}} St} \qquad \frac{}{F \triangleq^{\neg}_{\mathbf{lhs}} St}$$

$$\frac{F \not\sqsubseteq_{\mathbf{lhs}} \wr f \mid g \int_{\vec{x} \in ts} \quad \wr f \mid g \int_{\vec{x} \in ts} \triangleq^{\neg}_{\mathbf{lhs}} St}{\wr f \mid g \int_{\vec{x} \in ts} \triangleq^{\neg}_{\mathbf{lhs}} St, F} \qquad \frac{}{\wr f \mid g \int_{\vec{x} \in ts} \triangleq^{\neg}_{\mathbf{lhs}} \emptyset}$$

Subsumption: $F \sqsubseteq_{\mathbf{lhs}} \wr f \mid g \int_{\vec{x} \in ts}$ iff $F = \theta f$ and $\models \theta g$ for some $\theta = [\vec{t}/\vec{x}]$

Rewriting Runtime: Rewriting Semantics

- Unfolding rule body: $\overline{B} \ggg_{\textbf{rhs}} St$
  - Expands $\overline{B}$ into St

$$\frac{\overline{B} \ggg_{\textbf{rhs}} St \quad B \ggg_{\textbf{rhs}} St'}{\overline{B}, B \ggg_{\textbf{rhs}} St, St'} \qquad \frac{}{\varnothing \ggg_{\textbf{rhs}} \varnothing} \qquad \frac{}{F \ggg_{\textbf{rhs}} F}$$

$$\frac{\models [\vec{t}/\vec{x}]g \quad [t/\vec{x}]b \ggg_{\textbf{rhs}} F \quad \wr b \mid g \int_{\vec{x} \in ts} \ggg_{\textbf{rhs}} St}{\wr b \mid g \int_{\vec{x} \in \vec{t}, ts} \ggg_{\textbf{rhs}} F, St}$$

$$\frac{\not\models [\vec{t}/\vec{x}]g \quad \wr b \mid g \int_{\vec{x} \in ts} \ggg_{\textbf{rhs}} St}{\wr b \mid g \int_{\vec{x} \in \vec{t}, ts} \ggg_{\textbf{rhs}} St} \qquad \frac{}{\wr b \mid g \int_{\vec{x} \in \varnothing} \ggg_{\textbf{rhs}} \varnothing}$$

Rewriting Runtime: Rewriting Semantics

- Rewriting runtime transition: $\mathcal{P} \triangleright \langle St; \Psi \rangle \mapsto \langle St'; \Psi \rangle$
  - Applies a rule in $\mathcal{P}$ to transform St into St'

$$\frac{(\overline{H}_p \setminus \overline{H}_s \mid g \multimap \overline{B}) \in \mathcal{P} \qquad \models \theta g \\ \theta\overline{H}_p \triangleq_{\textbf{lhs}} St_p \quad \theta\overline{H}_s \triangleq_{\textbf{lhs}} St_s \quad \theta(\overline{H}_p, \overline{H}_s) \triangleq^{\neg}_{\textbf{lhs}} St \quad \theta\overline{B} \ggg_{\textbf{rhs}} St_b}{\mathcal{P} \triangleright \langle St_p, St_s, St; \Psi \rangle \mapsto \langle St_p, St_b, St; \Psi \rangle}$$

Application Runtime: Triggers and Actuators

- A local computation at location ℓ: $\langle \mathcal{A}; \psi \rangle \mapsto_\ell \langle \mathcal{T}; \psi' \rangle$
  - $\mathcal{A}$ is a set of actuator facts, introduced by the rewrite state St
  - $\mathcal{T}$ is a set of trigger facts, produced by the above local computation

$$\frac{\langle \mathcal{A}; \psi \rangle \mapsto_\ell \langle \mathcal{T}; \psi' \rangle}{\mathcal{P} \rhd \langle St, [\ell]\mathcal{A}; \Psi, [\ell]\psi \rangle \mapsto \langle St, [\ell]\mathcal{T}; \Psi, [\ell]\psi' \rangle}$$

- Entire computation must be happen at ℓ

Outline

Compilation of CoMingle Programs

System-centric specification

- High-level, concise
- Allows distributed events

```
rule pSwap :: [X]swap(Y,Z),
              {[X]item(I)|I->Is},
              {[Y]item(J)|J->Js},
              {[Z]item(K)|K->Ks}  --o [X]display(Msg,size(Js),Y), {[X]item(J)|J<-Js},
                                      [Y]display(Msg,size(Ks),Z), {[Y]item(K)|K<-Ks},
                                      [Z]display(Msg,size(Is),X), {[Z]item(I)|I<-Is}
                                      where Msg = "%s from %s".
```

Compilation of CoMingle Programs

System-centric specification

- High-level, concise
- Allows distributed events

```
rule pSwap :: [X]swap(Y,Z),
              {[X]item(I)|I->Is},
              {[Y]item(J)|J->Js},
              {[Z]item(K)|K->Ks}  --o [X]display(Msg,size(Js),Y), {[X]item(J)|J<-Js},
                                      [Y]display(Msg,size(Ks),Z), {[Y]item(K)|K<-Ks},
                                      [Z]display(Msg,size(Is),X), {[Z]item(I)|I<-Is}
                                      where Msg = "%s from %s".
```

Choreographic Transformation ↓ [Lam and Cervesato, 2013]

Node-centric specification

- Match facts within a node
- Handles lower-level concurrency
  - Synchronization
  - Progress
  - Atomicity and Isolation

```
rule pSwapTest :: -{ [X]inTrans__(_) }, [X]swap(Y,Z), {[X]item(I)|I -> Is} \ 1
                  --o exists T__. [X]inTrans__(T__), [Y]pSwapProbeY(T__,Y,Is,Z), [Z]pSwapProbeZ(T__,Y,Is,Z).

rule pSwapProbeY :: { [Y]inTrans__(P__)|P__->Ps__ }, {[Y]item(J)|J -> Js}
                 \ [Y]pSwapProbeY(T__,Y,Is,Z) | strongest(T__,Ps__) --o [X]pSwapReadyY(T__,Y).

rule pSwapProbeZ :: { [Z]inTrans__(P__)|P__->Ps__ }, {[Z]item(K)|K -> Ks}
                 \ [Z]pSwapProbeZ(T__,Y,Is,Z) | strongest(T__,Ps__) --o [X]pSwapReadyZ(T__,Z).

rule pSwapEngage :: [X]inTrans__(T__) \ [X]pSwapReadyY(T__,Y), [X]pSwapReadyZ(T__,Z) --o [X]pSwapInit(T__,Y,Z).

rule pSwapInit :: [X]pSwapInit(T__,Y,Z), [X]itemLock(), [X]swap(Y,Z), {[X]item(I)|I -> Is}
                  --o [X]pSwapLHSX(T__,Y,Is,Z), [Y]pSwapReqY(T__,X,Is,Z), [Z]pSwapReqZ(T__,X,Y,Is).

rule pSwapReqYSucc :: [Y]pSwapReqY(T__,X,Is,Z), [Y]itemLock(), {[Y]item(J)|J -> Js}
                  --o [Y]pSwapLHSY(T__,Y,Js).

rule pSwapReqZSucc :: [Z]pSwapReqZ(T__,X,Y,Is), [Z]itemLock(), {[Z]item(K)|K -> Ks}
                  --o [Z]pSwapLHSZ(T__,Z,Ks).

rule pSwapCommit :: [X]inTrans__(T__), [X]pSwapLHSX(T__,Y,Is,Z), [X]pSwapLHSY(T__,Y,Js), [X]pSwapLHSZ(T__,Z,Ks)
                 --o [X]display(Msg,lvar0,Y), {[X]item(J)|J -> Js},
                     [Y]display(Msg,lvar1,Z), {[Y]item(K)|K -> Ks},
                     [Z]display(Msg,lvar2,X), {[Z]item(I)|I -> Is},
                     [Y]itemLock(), [X]itemLock(), [Z]itemLock()
                    Msg = "%s from %s", lvar0 = (sizeJs), lvar1 = (sizeKs), lvar2 = (sizeIs)

rule pSwapReqYFail :: [Y]pSwapReqY(T__,X,Is,Z) --o [X]pSwapAbort(T__).

rule pSwapReqZFail :: [Z]pSwapReqZ(T__,X,Y,Is) --o [X]pSwapAbort(T__).

rule pSwapAbortX :: [X]pSwapAbort(T__) \ [X]inTrans__(T__), [X]pSwapLHSX(T__,Y,Is,Z)
                 --o [X]itemLock(), [X]swap(Y,Z), {[X]item(I)|I -> Is}.

rule pSwapAbortY :: [X]pSwapAbort(T__) \ [X]pSwapLHSY(T__,Y,Js)
                 --o [Y]itemLock(), {[Y]item(J)|J -> Js}.

rule pSwapAbortZ :: [X]pSwapAbort(T__) \ [X]pSwapLHSZ(T__,Z,Ks)
                 --o [Z]itemLock(), {[Z]item(K)|K -> Ks}.
```

Compilation of CoMingle Programs

System-centric specification
- High-level, concise
- Allows distributed events

```
rule pSwap :: [X]swap(Y,Z),
              {[X]item(I)|I->Is},
              {[Y]item(J)|J->Js},
              {[Z]item(K)|K->Ks}  --o [X]display(Msg,size(Js),Y), {[X]item(J)|J<-Js},
                                      [Y]display(Msg,size(Ks),Z), {[Y]item(K)|K<-Ks},
                                      [Z]display(Msg,size(Is),X), {[Z]item(I)|I<-Is}
                                      where Msg = "%s from %s".
```

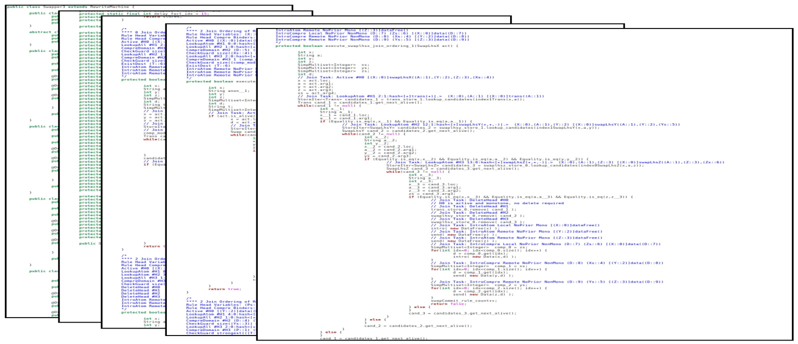
Choreographic Transformation ↓ [Lam and Cervesato, 2013]

Node-centric specification
- Match facts within a node
- Handles lower-level concurrency
  - Synchronization
  - Progress
  - Atomicity and Isolation

```
rule pSwapTest :: -{ [X]inTrans__(_) }, [X]swap(Y,Z), {[X]item(I)|I -> Is} \ 1
                   --o exists T__. [X]inTrans__(T__), [Y]pSwapProbeY(T__,Y,Is,Z), [Z]pSwapProbeZ(T__,Y,Is,Z).

rule pSwapProbeY :: { [Y]inTrans__(P__)|P__->Ps__ }, {[Y]item(J)|J -> Js}
                 \ [Y]pSwapProbeY(T__,Y,Is,Z) | strongest(T__,Ps__) --o [X]pSwapReadyY(T__,Y).

rule pSwapProbeZ :: { [Z]inTrans__(P__)|P__->Ps__ }, {[Z]item(K)|K -> Ks}
                 \ [Z]pSwapProbeZ(T__,Y,Is,Z) | strongest(T__,Ps__) --o [X]pSwapReadyZ(T__,Z).

rule pSwapEngage :: [X]inTrans__(T__) \ [X]pSwapReadyY(T__,Y), [X]pSwapReadyZ(T__,Z) --o [X]pSwapInit(T__,Y,Z).

rule pSwapInit :: [X]pSwapInit(T__,Y,Z), [X]itemLock(), [X]swap(Y,Z), {[X]item(I)|I -> Is}
                   --o [X]pSwapLHSX(T__,Y,Is,Z), [Y]pSwapReqY(T__,X,Is,Z), [Z]pSwapReqZ(T__,X,Y,Is).

rule pSwapReqYSucc :: [Y]pSwapReqY(T__,X,Is,Z), [Y]itemLock(), {[Y]item(J)|J -> Js}
                   --o [Y]pSwapLHSY(T__,Y,Js).

rule pSwapReqZSucc :: [Z]pSwapReqZ(T__,X,Y,Is), [Z]itemLock(), {[Z]item(K)|K -> Ks}
                   --o [Z]pSwapLHSZ(T__,Z,Ks).

rule pSwapCommit :: [X]inTrans__(T__), [X]pSwapLHSX(T__,Y,Is,Z), [X]pSwapLHSY(T__,Y,Js), [X]pSwapLHSZ(T__,Z,Ks)
                 --o [X]display(Msg,lvar0,Y), {[X]item(J)|J -> Js},
                     [Y]display(Msg,lvar1,Z), {[Y]item(K)|K -> Ks},
                     [Z]display(Msg,lvar2,X), {[Z]item(I)|I -> Is},
                     [Y]itemLock(), [X]itemLock(), [Z]itemLock()
                    Msg = "%s from %s", lvar0 = (sizeJs), lvar1 = (sizeKs), lvar2 = (sizeIs)

rule pSwapReqYFail :: [Y]pSwapReqY(T__,X,Is,Z) --o [X]pSwapAbort(T__).

rule pSwapReqZFail :: [Z]pSwapReqZ(T__,X,Y,Is) --o [X]pSwapAbort(T__).

rule pSwapAbortX :: [X]pSwapAbort(T__) \ [X]inTrans__(T__), [X]pSwapLHSX(T__,Y,Is,Z)
                 --o [X]itemLock(), [X]swap(Y,Z), {[X]item(I)|I -> Is}.

rule pSwapAbortY :: [X]pSwapAbort(T__) \ [X]pSwapLHSY(T__,Y,Js)
                 --o [Y]itemLock(), {[Y]item(J)|J -> Js}.

rule pSwapAbortZ :: [X]pSwapAbort(T__) \ [X]pSwapLHSZ(T__,Z,Ks)
                 --o [Z]itemLock(), {[Z]item(K)|K -> Ks}.
```

Imperative Compilation ↓ [Lam and Cervesato, 2014]

Low-level imperative compilation
- Java code
- Low-level network calls
- Operationalize multiset rewriting
- Trigger and actuator interfaces



Outline

Implementation

- Prototype Available at

  `https://github.com/sllam/comingle`

- Networking via Wifi-Direct
- More backends coming soon (Android Beam, Bluetooth)
- Proof-of-concept Apps
  - Drag Racing
  - Battleships
  - P2P Wifi-Direct Directory
  - Swarbble
- See tech.report [Lam and Cervesato, 2015] for details!

Drag Racing

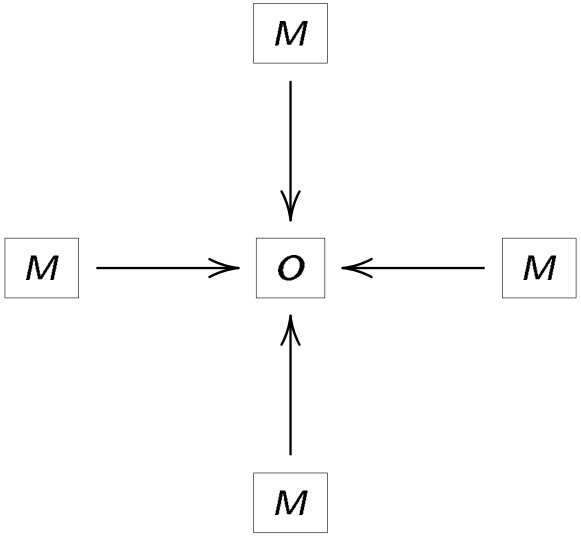
- Inspired by Chrome Racer (www.chrome.com/racer)
- Race across a group of mobile devices
- Decentralized communication (over Wifi-Direct)

Implementing Drag Racing in CoMingle

```
rule init :: [I]initRace(Ls)
   --o {[A]next(B)|(A,B)<-Cs}, [E]last(),
       {[I]has(P), [P]all(Ps), [P]at(I), [P]rendTrack(Ls) | P<-Ps}
       where (Cs,E) = makeChain(I,Ls), Ps = list2mset(Ls).

rule start :: [X]all(Ps) \ [X]startRace() --o {[P]release()|P<-Ps}.

rule tap   :: [X]at(Y) \ [X]sendTap() --o [Y]recvTap(X).

rule trans :: [X]next(Z) \ [X]exiting(Y), [Y]at(X) --o [Z]has(Y), [Y]at(Z).

rule win   :: [X]last() \ [X]all(Ps), [X]exiting(Y) --o {[P]decWinner(Y) | P <- Ps}.
```

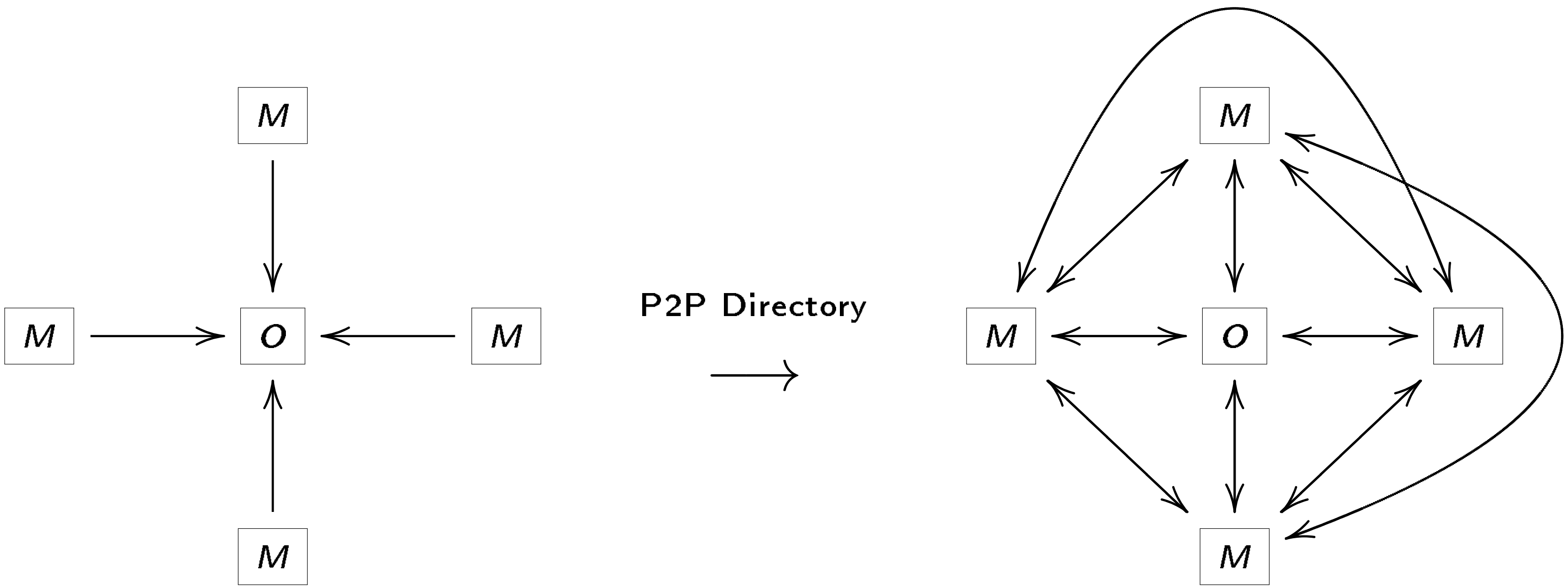
- + 862 lines of properly indented Java code
  - 700++ lines of local operations (e.g., display and UI operations)
  - $<$ 100 lines for initializing CoMingle runtime

Wifi P2P Directory



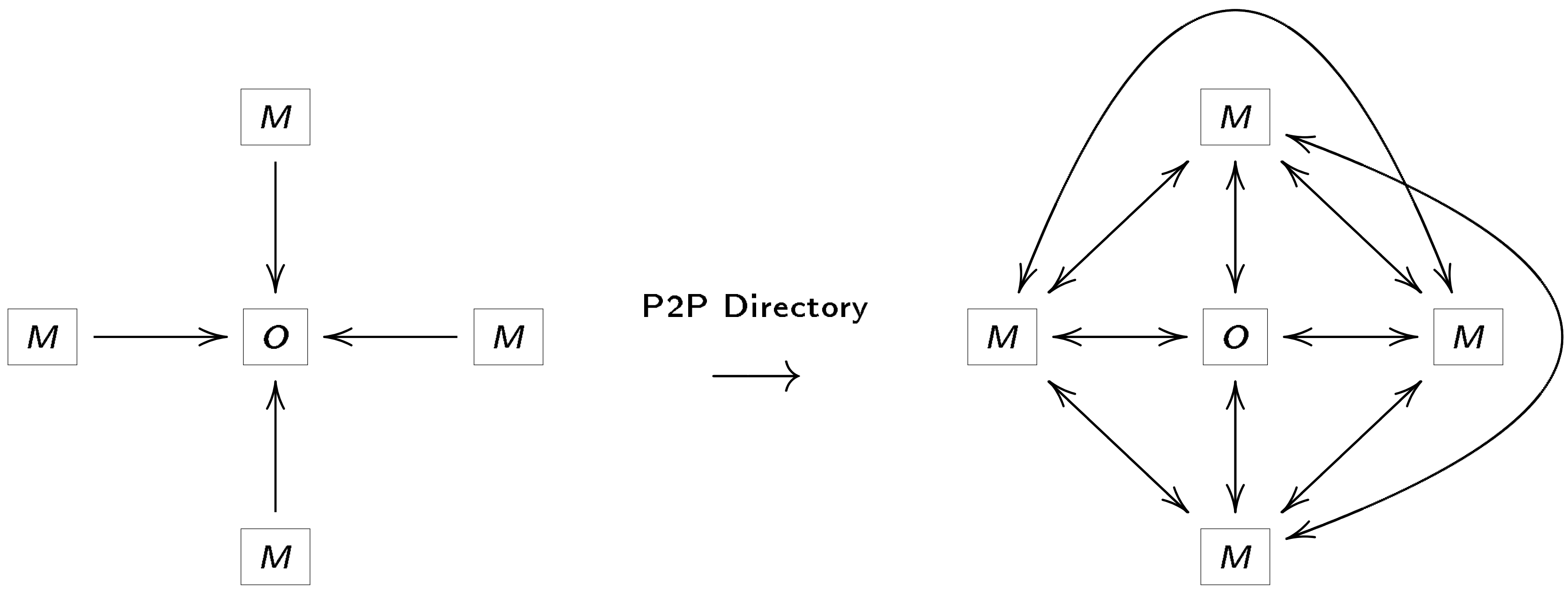
- Wifi Direct APIs in the Android SDK
  - Enable "easy" setup of a mobile ad-hoc network
  - One device act as the *owner* (O)
  - Others are *members* (M)
  - But only takes you half-way: Each M has IP of O only

Wifi P2P Directory



- Wifi P2P Directory program
  - Maintains a dynamic IP directory on each node
  - Implements a daemon on each M to receive updates from O
  - Implements a daemon on O that broadcasts updates to each M

Wifi P2P Directory



- Implemented in CoMingle within each CoMingle App
  - P2P Directory bootstrapped into CoMingle initialization
  - Runs in the background as a separate CoMingle runtime instance

Implementing P2P Directory in CoMingle

```
rule owner  :: [O]startOwner(C)  --o [O]owner(C), [O]joined(O).
rule member :: [M]startMember(C) --o [M]member(C).

rule connect :: [M]member(C) \ [M]connect(N) --o [O]joinRequest(C,N,M)
                                                  where O = ownerLoc().

rule join :: [O]owner(C), {[O]joined(M')|M'->Ms} \ [O]joinRequest(C,N,M) | notIn(M,Ms)
                    --o {[M']added(D)|M'<-Ms}, {[M]added(D')|D'<-Ds},
                        [M]added(D), [O]joined(M), [M]connected()
                        where IP = lookupIP(M), D = (M,IP,N), Ds = retrieveDir().

rule quitO :: [O]owner(C), [O]quit(), {[O]joined(M)|M->Ms} --o {[M]ownerQuit()|M<-Ms} .

rule quitM :: {[O]joined(M')|M'->Ms.not(M' = M)} \ [M]member(C), [M]quit(), [O]joined(M)
                    --o {[M']removed(M)|M'<-Ms}, [M]deleteDir().
```

- Two implementations of P2P Directory
  - "Vanilla" Java + Android SDK: 694 lines of Java code
  - CoMingle + Java + Android SDK: 53 lines of CoMingle code + 154 lines of Java code
- All code is properly indented
- Omitting common libraries used by both implementations

Outline

Conclusion

- CoMingle: Distributed logic programming language
  - For programming distributed mobile applications
  - Based on decentralized multiset rewriting with comprehension patterns
- Prototype implementation
  - Available at `https://github.com/sllam/comingle`
  - Example apps available for download as well
  - Show your support, please STAR CoMingle GitHub repository!

Future Work

- Front end refinements
  - Additional primitive types
  - More syntactic sugar
  - Refine Java interfaces
- Incremental extensions
  - Additional networking middlewares (Bluetooth, Android Beam, Wifi)
  - Sensor abstraction in CoMingle (e.g., GPS, speedometer)
  - More platforms (iOS, Raspberry Pi, Arduino, backend servers)
- Going beyond toy applications
  - Augmenting event/conference applications
  - Social interactive mobile applications

Questions?

Bibliography

Cruz, F., Rocha, R., Goldstein, S. C., and Pfenning, F. (2014).
A linear logic programming language for concurrent programming over graph structures.
CoRR, abs/1405.3556.

Frühwirth, T. and Raiser, F. (2011).
Constraint Handling Rules: Compilation, Execution and Analysis.
ISBN 9783839115916. BOD.

Lam, E. and Cervesato, I. (2013).
Decentralized Execution of Constraint Handling Rules for Ensembles.
In *PPDP'13*, pages 205–216, Madrid, Spain.

Lam, E. and Cervesato, I. (2014).
Optimized Compilation of Multiset Rewriting with Comprehensions.
In *APLAS'14*, pages 19–38. Springer LNCS 8858.

Lam, E. and Cervesato, I. (2015).
Comingle: Distributed Logic Programming for Decentralized Android Applications.
Technical Report CMU-CS-15-101, Carnegie Mellon University.