Comingle: Distributed Logic Programming for Decentralized Mobile Ensembles

Edmund Soon Lee Lam(✉), Iliano Cervesato, and Nabeeha Fatima

Carnegie Mellon University, University in Pittsburgh, Pennsylvania, Pittsburgh, PA, USA
{sllam,nhaque}@andrew.cmu.edu, iliano@cmu.edu

Abstract. Comingle is a logic programming framework aimed at simplifying the development of applications distributed over multiple mobile devices. Applications are written as a single declarative program (in a system-centric way) rather than in the traditional node-centric manner, where separate communicating code is written for each participating node. Comingle is based on committed-choice multiset rewriting and is founded on linear logic. We describe a prototype targeting the Android operating system and illustrate how Comingle is used to program distributed mobile applications. As a proof of concept, we discuss several such applications orchestrated using Comingle.

1 Introduction

Distributed computing, the coordination of independent computations to achieve a desired objective, has become one of the defining technologies of modern society. We rely on it every time we use a search engine like Google, every time we make a purchase on Amazon, in fact every time we use the Internet. In recent years, *mobile* distributed computing has taken off thanks to advances in mobile technologies, from inexpensive sensors and low-energy wireless links to the very smartphones we carry around: apps talk to each other both within a phone and across phones, connected devices work together to make our homes safer and more comfortable, and personal health monitors combine sensor data into a picture of our well-being. Each such system constitutes a decentralized mobile application which orchestrates the computations of its various constituent nodes. As such applications gain in sophistication, it becomes harder to ensure that they correctly and reliably deliver the desired behavior using traditional programming models. Specifically, writing separate communicating programs for each participating node becomes more costly and error-prone as the need for node-to-node coordination grows.

In this paper, we introduce Comingle, a framework aimed at simplifying the development of distributed applications over a decentralized ensemble of mobile devices. Comingle supports a system-centric style of programming, where the distributed behavior of an application is expressed as a single program, rather than the traditional

This work was made possible by grant JSREP 4-003-2-001, *Effective Parallel and Distributed Programming via Join Pattern with Guards, Propagation and More*, from the Qatar National Research Fund (a member of the Qatar Foundation). The statements made herein are solely the responsibility of the authors.

T. Holvoet and M. Viroli (Eds.): COORDINATION 2015, LNCS 9037, pp. 51–66, 2015.
DOI: 10.1007/978-3-319-19282-6_4

node-centric style mentioned above. This system-centric view underlies popular frameworks such as Google Web Toolkit [7] (for client-server web development) and Map Reduce [4] (for parallel distributed algorithms on large-scale computing clusters). In earlier work [9,10], we generalized this approach to a much broader class of distributed computations by relying on a form of logic programming to orchestrate interactive distributed computations [9,10]. Comingle specializes this work to distributed applications running on mobile devices. Comingle is based on committed-choice multiset rewriting extended with explicit locality [9] and multiset comprehension patterns [10]. This provides declarative and concise means of implementing distributed computations, thus allowing the programmer to focus on *what* computations to synchronize rather than *how* to synchronize them. The present work extends [9] by introducing *triggers* and *actuators* to integrate the Comingle multiset rewriting runtime with traditional code from mainstream mobile development frameworks (specifically Java and the Android SDK). This allows a developer to marry the best of both programming paradigms, using Comingle to orchestrate distributed computations among devices and traditional native code for computations within a device (e.g., user interface functionalities, local computations). The main contributions of this paper are as follows:

- We detail the semantics of Comingle, in particular the use of triggers and actuators as an abstract interface between Comingle and a device's application runtime.
- We describe a prototype implementation of Comingle, a runtime system implemented in Java and integrated with the Android SDK.
- As a proof of concept, we show three case-studies of distributed applications orchestrated by Comingle on the Android SDK.

The rest of the paper is organized as follows: we illustrate Comingle by means of an example in Section 2. In Section 3, we introduce its abstract syntax and its semantics, while Section 4 outlines our compiler and runtime system for the Android SDK. In Section 5, we examine three case-study applications implemented in Comingle. We discuss related works in Section 6 and make some concluding remarks in Section 7. Further details can be found in a companion technical report [12].

2 A Motivating Example

Figure 1 shows a simple Comingle program that lets two generic devices swap data that they each possess on the basis of a pivot value P and displays on each of them the number of items swapped, all in one atomic step. This program gives a bird eye's view of the exchanges that need to take place — it is system-centric. Our prototype will then compile it into the node-centric code that needs to run at each device to realize this behavior. The high-level Comingle program in Figure 1 relies on a few functionalities expressed using the devices' native programming support (Java and the Android SDK in our case). Specifically, these functionalities are the two local functions, `size` and `format`, imported in lines `1-4`, and the code associated with triggers and actuators (see below). This low-level code (not shown) implements purely local computations.

In Comingle, devices are identified by means of a *location* and a piece of information held at location ℓ is represented as a *located fact* of the form $[\ell]\, p(\vec{t})$ where p is

```
1  module comingle.lib.ExtLib import {
2          size   :: A -> int,
3          format :: (string,A) -> string
4  }
5
6  predicate swap    :: (loc,int) -> trigger.
7  predicate item    :: int -> fact.
8  predicate display :: string -> actuator.
9
10 rule pSwap :: [X]swap(Y,P),
11               { [X]item(I) | I -> Is. I <= P },
12               { [Y]item(J) | J -> Js. J >= P }
13      --o [X]display( format("Received %s items from %s", (size(Js),Y)) ),
14          [Y]display( format("Received %s items from %s", (size(Is),X)) ),
15          { [X]item(J) | J <- Js }, { [Y]item(I) | I <- Is }.
```

Fig. 1. Pivot Swap, orchestrated by Comingle

a predicate name and $\vec{t}$ are terms. The program in Figure 1 mentions two generic locations, `X` and `Y`, and uses the three predicates declared on lines `6-8`. A located fact of the form $[\ell]\texttt{swap}(\ell', P)$ represents ℓ's intent to swap data with device ℓ' based on the pivot value P, fact $[\ell]\texttt{item}(I)$ indicates that value I is held at location ℓ, while $[\ell]\texttt{display}(S)$ represents a message S to be shown on ℓ's screen. From a system-centric perspective, the set of all located facts defines the *rewriting state* of the system. The rewriting state evolves through the application of Comingle rules and indirectly by the effect of the local computation of each device.

Lines `10-15` in Figure 1 define a Comingle rule called `pSwap`. We call the comma-separated expressions before "`--o`" the rule *heads*, while the expressions after it are collectively called its *body*. Informally, applying a Comingle rule to the current state rewrites an instance of its head into the corresponding instance of its body. Rule heads and body can contain parametric facts such as `[X]swap(Y,P)`, where `X`, `Y` and `P` are variables, and *comprehension patterns* which stand for a multiset of facts in the rewriting state. In our example, the comprehension pattern {`[X]item(I) | I -> Is. I <= P`} identifies all of `X`'s items `I` such that `I <= P`. Similarly, all of `Y`'s items `J` such that `J >= P` are identified by {`[Y]item(J) | J -> Js. J >= P`}. The instances of `I` and `J` matched by each comprehension pattern are accumulated in the variables `Is` and `Js`, respectively. Finally, these collected bindings are used in the rule body to complete the rewriting by redistributing all of `X`'s selected data to `Y` and vice versa, as well as invoking the appropriate display messages on `X`'s and `Y`'s screen.

Facts such as $\texttt{item}(I)$ are meaningful only at the rewriting level. Facts are also used as an interface to a device's local computations. Specifically, facts like $[\ell]\texttt{swap}(\ell', P)$ are entered into the rewriting state by a local program running at ℓ and used to trigger rule applications. These *trigger facts*, which we underline as $[\ell]\underline{\texttt{swap}}(\ell', P)$ for emphasis, are only allowed in the heads of a rule. Dually, facts like $[\ell]\texttt{display}(S)$ are generated by the rewriting process for the purpose of starting a local computation at ℓ, here displaying a message on ℓ's screen. This is an *actuator fact*, which we underline with a dashed line, as in $[\ell]\underline{\texttt{display}}(S)$, for clarity. Each actuator predicate is associated with a local function which is invoked when the rewriting engine deposits an instance in the state (actuators can appear only in a rule body). For example, actuators

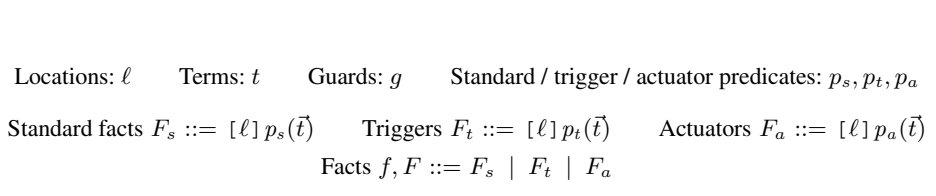

Fig. 2. Abstract Syntax and Runtime Artifacts of Comingle

of the form $[\ell]\,\texttt{display}(S)$ are concretely implemented using a Java callback operation (not shown here) that calls the Android SDK's `toast` pop-up notification library to display the message S on ℓ's screen. This callback is invoked at ℓ every time the Comingle runtime produces an instance $[\ell]\,\texttt{display}(S)$.

By being system-centric, the code in Figure 1 lets the developer think in terms of overall behavior rather than reason from the point of view of each device, delegating to the compiler to deal with communication and synchronization, two particularly error-prone aspects of distributed computing. This also enable global type-checking and other forms of static validation, which are harder to achieve when writing separate programs. This code is also declarative, which simplifies reasoning about its correctness and security. Finally, this code is concise: just 15 lines. A native implementation of this example, while not difficult, is much longer.

3 Abstract Syntax and Semantics

In this section, we describe the abstract semantics of Comingle. We begin by first introducing the notations used throughout this section. We write $\overline{o}$ for a multiset of syntactic objects o. We denote the extension of a multiset $\overline{o}$ with an object o as "$\overline{o}, o$", with $\varnothing$ indicating the empty multiset. We also write "$\overline{o}_1, \overline{o}_2$" for the union of multisets $\overline{o}_1$ and $\overline{o}_2$. We write $\vec{o}$ for a tuple of o's and $[\vec{t}/\vec{x}]o$ for the simultaneous replacement within object o of all occurrences of variable x_i in $\vec{x}$ with the corresponding term t_i in $\vec{t}$. When traversing a binding construct (e.g., a comprehension pattern), substitution implicitly α-renames variables as needed to avoid capture. It will be convenient to assume that terms get normalized during substitution.

3.1 Abstract Syntax

The top part of Figure 2 defines the abstract syntax of Comingle. The concrete syntax used in the various examples in this paper maps to this abstract syntax. *Locations* ℓ are names that uniquely identify computing nodes, and the set of all nodes participating in a Comingle computation is called an *ensemble*. At the Comingle level, computation happens by rewriting *located facts* F of the form $[\ell]\, p(\vec{t})$. We categorize predicate names

p into *standard*, *trigger* and *actuator*, indicating them with p_s, p_r and p_a, respectively. This induces a classification of facts into standard, trigger and actuator facts, denoted F_s, F_t and F_a, respectively. Facts also carry a tuple $\vec{t}$ of *terms*. The abstract semantics of Comingle is largely agnostic to the specific language of terms.

Computation in Comingle happens by applying *rules* of the form $\overline{H}_p \setminus \overline{H}_s \mid g \multimap \overline{B}$. We refer to $\overline{H}_p$ and $\overline{H}_s$ as the *preserved* and the *consumed head* of the rule, to g as its *guard* and to $\overline{B}$ as its *body*. The heads and the body of a rule consist of *atoms* f and of *comprehension patterns* of the form $\wr f \mid g \int_{\vec{x} \in t}$. An atom f is a located fact $[\ell]\, p(\vec{t})$ that may contain variables in the terms $\vec{t}$ or even as the location ℓ. Atoms in rule heads are either standard or trigger facts (F_s or F_t), while atoms in a rule body are standard or actuator facts (F_s or F_t). Guards in rules and comprehensions are Boolean-valued expressions constructed from terms and are used to constrain the values that the variables in a rule can assume. Just like for terms we keep guards abstract, writing $\models g$ to express that ground guard g is satisfiable. A comprehension pattern $\wr f \mid g \int_{\vec{x} \in t}$ represents a multiset of facts that match the atom f and satisfy guard g under the bindings of variables $\vec{x}$ that range over t, a multiset of tuples called the *comprehension range*. The scope of $\vec{x}$ is the atom f and the guard g. We implicitly α-rename bound variables to avoid capture. Abstractly, a Comingle *program* is a collection of rules.

The concrete syntax of Comingle is significantly more liberal than what we just described. In particular, components $\overline{H}_p$ and g can be omitted if empty. We concretely write a comprehension pattern $\wr f \mid g \int_{\vec{x} \in t}$ as $\{f \mid \vec{x} \texttt{ -> } t\texttt{. } g\}$ in rule heads and $\{f \mid \vec{x} \texttt{ <- } t\texttt{. } g\}$ in a rule body, where the direction of the arrow acts as a reminder of the flow of information. Terms in the current prototype include standard base types such as integers and strings, locations, term-level multisets, and lists. Its guards are relations over such terms (e.g., equality and $x < y$) and can contain effect-free operations imported from the local application (e.g., `size` and `format` in Figure 1).

3.2 Abstract Semantics

We will describe the computation of a Comingle system by means of a small-step transition semantics. Its basic judgment will have the form $\mathcal{P} \rhd \Theta \mapsto \Theta'$ where $\mathcal{P}$ is a program, Θ is a *state* and Θ' is a state that can be reached in one (abstract) step of computation. A state Θ has the form $\langle St; \Psi \rangle$. The first component St is a collection of ground located facts $[\ell]\, p(\vec{t})$ and is called the *rewriting state* of the system. Comingle rules operate exclusively on the rewriting state. The second component, the *application state* Ψ, is the collection of the *local states* $[\ell]\psi$ of each computing node ℓ and captures the notion of state of the underlying computation model (the Java virtual machine in our Android-based prototype) — it typically has nothing to do with facts. As we will see, a local computation step transforms the application state Ψ but can also consume triggers from the rewriting state and add actuators into it. These run-time artifacts are formally defined at the bottom of Figure 2.

We will now describe the two types of state transitions $\mathcal{P} \rhd \Theta \mapsto \Theta'$ in Comingle: the application of a rule and a local step — see Figure 5 for a preview.

Rewriting Steps. The application of a Comingle rule $\overline{H}_p \setminus \overline{H}_s \mid g \multimap \overline{B}$ involves two main operations: identifying fragments of the rewriting state St that match the

Matching: $\overline{H} \triangleq_{\mathbf{lhs}} St \qquad H \triangleq_{\mathbf{lhs}} St$

$$\frac{\overline{H} \triangleq_{\mathbf{lhs}} St \quad H \triangleq_{\mathbf{lhs}} St'}{\overline{H}, H \triangleq_{\mathbf{lhs}} St, St'}\,(\mathsf{I}_{mset\text{-}1}) \qquad \frac{}{\varnothing \triangleq_{\mathbf{lhs}} \varnothing}\,(\mathsf{I}_{mset\text{-}2}) \qquad \frac{}{F \triangleq_{\mathbf{lhs}} F}\,(\mathsf{I}_{fact})$$

$$\frac{[\vec{t}/\vec{x}]f \triangleq_{\mathbf{lhs}} F \quad \models [\vec{t}/\vec{x}]g \quad \wr f \mid g\wr_{\vec{x}\in\overline{ts}} \triangleq_{\mathbf{lhs}} St}{\wr f \mid g\wr_{\vec{x}\in\vec{t},\overline{ts}} \triangleq_{\mathbf{lhs}} St, F}\,(\mathsf{I}_{comp\text{-}1}) \qquad \frac{}{\wr f \mid g\wr_{\vec{x}\in\varnothing} \triangleq_{\mathbf{lhs}} \varnothing}\,(\mathsf{I}_{comp\text{-}2})$$

Residual Non-matching: $\overline{H} \triangleq^{\neg}_{\mathbf{lhs}} St \qquad H \triangleq^{\neg}_{\mathbf{lhs}} St$

$$\frac{\overline{H} \triangleq^{\neg}_{\mathbf{lhs}} St \quad H \triangleq^{\neg}_{\mathbf{lhs}} St}{\overline{H}, H \triangleq^{\neg}_{\mathbf{lhs}} St}\,(\mathsf{I}^{\neg}_{mset\text{-}1}) \qquad \frac{}{\varnothing \triangleq^{\neg}_{\mathbf{lhs}} St}\,(\mathsf{I}^{\neg}_{mset\text{-}2}) \qquad \frac{}{F \triangleq^{\neg}_{\mathbf{lhs}} St}\,(\mathsf{I}^{\neg}_{fact})$$

$$\frac{F \not\sqsubseteq_{\mathbf{lhs}} \wr f \mid g\wr_{\vec{x}\in ts} \quad \wr f \mid g\wr_{\vec{x}\in ts} \triangleq^{\neg}_{\mathbf{lhs}} St}{\wr f \mid g\wr_{\vec{x}\in ts} \triangleq^{\neg}_{\mathbf{lhs}} St, F}\,(\mathsf{I}^{\neg}_{comp\text{-}1}) \qquad \frac{}{\wr f \mid g\wr_{\vec{x}\in ts} \triangleq^{\neg}_{\mathbf{lhs}} \varnothing}\,(\mathsf{I}^{\neg}_{comp\text{-}2})$$

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

Fig. 3. Matching a Rule Head

rule heads $\overline{H}_p$ and $\overline{H}_s$, and replacing $\overline{H}_s$ in the rewriting state with the corresponding instance of the body $\overline{B}$. We now review how these operations are formalized in the presence of comprehension patterns and then describe how they are combined during a rewriting step (taking the guard g into account). Further details can be found in [10].

Matching Rule Heads. Let $\overline{H}$ be a (preserved or consumed) rule head without free variables — we will deal with the more general case momentarily. Intuitively, matching $\overline{H}$ against a store St means splitting St into two parts, St^+ and St^-, and checking that $\overline{H}$ matches St^+ completely. The latter is achieved by the judgment $\overline{H} \triangleq_{\mathbf{lhs}} St^+$ defined in the top part of Figure 3. Rules $\mathsf{I}_{mset\text{-}*}$ partition St^+ into fragments to be matched by each atom in $\overline{H}$: plain facts F must occur identically (rule I_{fact}) while for comprehension atoms $\wr f \mid g\wr_{\vec{x}\in\overline{ts}}$ the state fragment must contain a distinct instance of f for every element of the comprehension range $\overline{ts}$ that satisfies the comprehension guard g (rules $\mathsf{I}_{comp\text{-}*}$).

In Comingle, comprehension patterns must match *maximal* fragments of the rewriting state. Therefore, no comprehension pattern should match any fact in St^-. This check is captured by the judgment $\overline{H} \triangleq^{\neg}_{\mathbf{lhs}} St^-$ in the bottom part of Figure 3. Rules $\mathsf{I}^{\neg}_{mset\text{-}*}$ tests each individual atom and rule $\mathsf{I}^{\neg}_{fact}$ ignore facts. Rules $\mathsf{I}^{\neg}_{comp\text{-}*}$ deal with comprehensions $\wr f \mid g\wr_{\vec{x}\in\overline{ts}}$: they check that no fact in St^- matches any instance of f while satisfying g — note that the comprehension range $\overline{ts}$ is not taken into account.

Processing Rule Bodies. Applying a Comingle rule involves extending the rewriting state with the facts corresponding to its body. This operation is specified in Figure 4 for a closed body $\overline{B}$. Rules $\mathsf{r}_{mset\text{-}*}$ go through $\overline{B}$. Atomic facts F are added immediately (rule r_{fact}). Instead, comprehension atoms $\wr f \mid g\wr_{\vec{x}\in\overline{ts}}$ need to be *unfolded* (rules $\mathsf{r}_{comp\text{-}*}$):

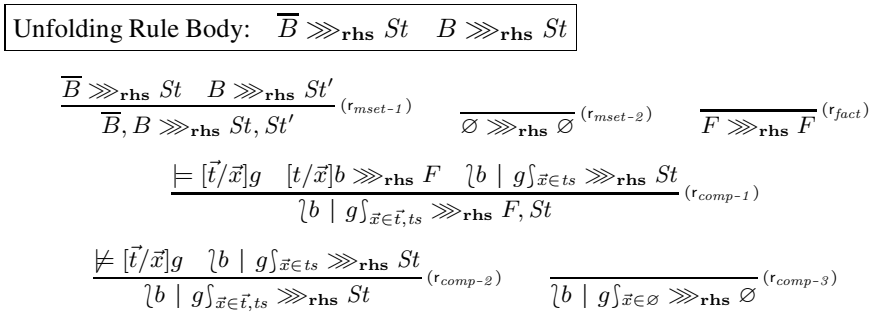

Fig. 4. Processing a Rule Body

for every item $\vec{t}$ in $\overline{ts}$ that satisfies the guard g, the corresponding instance $[\vec{t}/\vec{x}]f$ is added to the rewriting state; instances that do not satisfy g are discarded.

Rule Application. Rule rw_ens in Figure 5 brings these ingredients together and describes a step of computation that applies a rule $\overline{H}_p \setminus \overline{H}_s \mid g \multimap \overline{B}$. This involves identifying a closed instance of the rule obtained by means of a substitution θ. The instantiated guard must be satisfiable ($\models \theta g$) and we must be able to partition the rewriting state into three parts St_p, St_s and St. The instances of the preserved and consumed heads must match fragments St_p and St_s respectively ($\theta\overline{H}_p \triangleq_{\mathbf{lhs}} St_p$ and $\theta\overline{H}_s \triangleq_{\mathbf{lhs}} St_s$), while the remaining fragment St must be free of residual matchings ($(\theta(\overline{H}_p, \overline{H}_s) \triangleq^{\neg}_{\mathbf{lhs}} St$). The rule body instance $\theta\overline{B}$ is then unfolded ($\theta\overline{B} \ggg_{\mathbf{rhs}} St_b$) into St_b which replaces $\overline{St}_s$ in the rewriting state.

Rule rw_ens embodies a system-centric abstraction of the rewriting semantics of Comingle as it atomically accesses facts at arbitrary locations. Indeed, it views the facts of all participating locations in the ensemble as one virtual collection. Our prototype, discussed in Section 4, is instead based on a concurrent, node-centric model of computation, where each node manipulates its local facts and exchanges message with other nodes. We achieve this by compiling Comingle rules into the code that runs at each participating node [9].

Local Steps. Global rewriting steps can be interleaved by local computations at any node ℓ. From the point of view of Comingle, such local computations are viewed as an abstract transition $\langle \mathcal{A}; \psi \rangle \mapsto_l \langle \psi'; \mathcal{T} \rangle$ that consumes some actuators $\mathcal{A}$ located at ℓ, modifies ℓ's internal application state ψ into ψ', and produces some triggers $\mathcal{T}$. Note that an abstract transition of this kind can (and generally will) correspond to a large number of steps of the underlying model of computation of node ℓ. Rule rw_loc in Figure 5 incorporate local computation into the abstract semantics of Comingle. Here, we write $[\ell]\mathcal{A}$ for a portion of the actuators located at ℓ in the current rewriting state — there may be others. We similarly write $[\ell]\mathcal{T}$ for the action of locating each trigger in $\mathcal{T}$ at ℓ.

Rule rw_loc enforces locality by drawing actuators strictly from ℓ and putting back triggers at ℓ. In particular, local computations at a node cannot interact with other nodes.

Local transitions: $\langle \mathcal{A}; \psi \rangle \mapsto_l \langle \mathcal{T}; \psi' \rangle$

Comingle transitions: $\mathcal{P} \rhd \langle St; \Psi \rangle \mapsto \langle St; \Psi \rangle$

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

$$\frac{\langle \mathcal{A}; \psi \rangle \mapsto_l \langle \mathcal{T}; \psi' \rangle}{\mathcal{P} \rhd \langle St, [l]\mathcal{A}; \Psi, [l]\psi \rangle \mapsto \langle St, [l]\mathcal{T}; \Psi, [l]\psi' \rangle} \text{(rw\_loc)}$$

Fig. 5. Abstract Semantics of Comingle

Hence, communication and orchestration can only occur through rewriting steps, defined by rule rw_ens. Note also that, since local transitions are kept abstract and are parametrized by a location, rule rw_loc accommodates ensembles that comprise devices based on different underlying models of computation.

4 Implementation

We now describe our Comingle prototype. In Section 4.1, we highlight the compilation phase, while Section 4.2 discusses the runtime system. Source code and examples are available for download at https://github.com/sllam/comingle.

4.1 Compilation

The Comingle front-end compiler consists of a typical lexer and parser, type-checker, an intermediate language preprocessor and a code generator, all implemented in Python. The type-checker enforces basic static typing of Comingle programs via a constraint solving approach adapted from [14] that allows for concise syntax highlight of type error sites. This is achieved by having the type-checker generate typing constraints with additional bookkeeping data to pinpoint the syntax fragments responsible for each error. Satisfiability of these typing constraints are determined by an SMT solver library built on top of Microsoft's Z3 [3]. Our SMT solver library includes an extension to reason about set comprehensions [11] which we use for optimizations involving comprehension patterns. An example is the selection of the indexing structures used by the Comingle runtime to carry out multiset matching with the best possible asymptotic time complexity [10]. Once a program has been statically checked, the compiler first applies a high-level source-to-source transformation [9] that converts a class of system-centric Comingle programs into node-centric rules. In addition to preserving soundness, the resulting node-centric program explicitly implements the communications and synchronizations that are required to correctly orchestrate the distributed execution of multi-party Comingle rules among a group of participating devices. Details of this choreographic transformation are out of the scope of this paper, but can be found in [9]. Finally, the code generator produces Java code that implements multiset matching as specified by the node-centric encodings. This generated matching code uses a

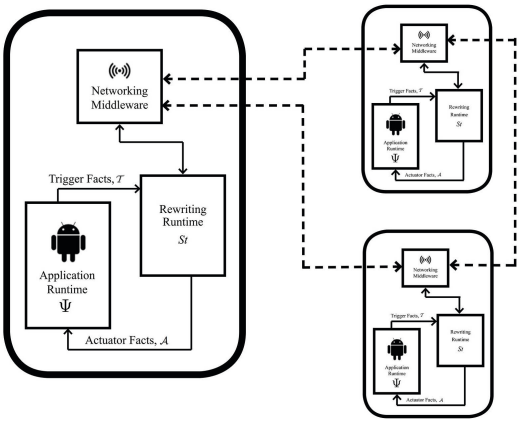

Fig. 6. Runtime System of a Distributed Comingle Mobile Application

compilation scheme formalized in [10] that first compiles node-centric code into a sequence of procedural operations, each of which implements a part of the matching and unfolding operations described in Section 3.2.

4.2 Runtime System

Figure 6 illustrates the organization of a running Comingle ensemble. Within each mobile device, the Comingle runtime has three components: a *rewriting runtime* that executes compiled rewrite rules, an *application runtime* that performs all local operations on the mobile device, and a *network middleware* that provides the basic communication primitives between the mobile devices. In the rest of this section, we highlight the important features of each of these components.

Rewriting Runtime. The rewriting runtime implements an operational semantics [10] which is sound with respect to the abstract semantics highlighted in Section 3. This operational semantics implements rule rw_ens on the node-centric rewriting rules resulting from the compilation process. In particular, it performs matching by incrementally processing atoms in a rule head on the basis of *newly added* facts. This execution model is highly compatible with our setup, where multiset rewriting is driven by external triggers generated by the local application runtime. Facts are matched to rule heads in top-down, left-to-right order, while facts in a rule body are processed left to right. The actions associated with actuators are executed in order of rule application. Each instance of the rewriting runtime is single-threaded, which entails that actuations invoked on the same device are guaranteed to be sequentially consistent with respect to the local ordering of rule application.

The rewriting runtime is implemented as a set of Java libraries. During compilation, the code generator produces Java code sprinkled with calls to functions from these libraries. Matching, for instance, is realized through various library functions that manipulate the data structures that implement the rewriting state St, supporting multi-index storage for efficiently querying facts. Communication is realized by other library calls that interface with the network middleware to send and receive facts to and from

other participating instances of Comingle. Other library functions allows the rewriting runtime to call actuators that affects the local application runtime. Furthermore, the rewriting runtime exposes interface functions to the local application to carry out administrative commands (e.g., start or stop rewriting) as well as interfaces to add user-defined triggering facts to the rewriting runtime. These interface functions, called by the rewriting runtime, are engineered to be abstract and they make no assumptions about the local application calling them, and hence can be customized for various platforms.

Application Runtime. The application runtime is the Android application that implements rule rw_loc, performing all the local operations on the mobile device, from screen rendering to managing callback routines invoked by user input (e.g., keystrokes, taps on the display). It is implemented in Java with the Android SDK, but also uses a library (distributed as part of Comingle) that concretizes the interface functions that the rewriting runtime invokes. Its purpose is to allow the application developer to integrate locally-defined functions into Comingle rewriting rules (as shown in Figure 1). Specifically, it includes a set of predefined actuation callback methods for the Android SDK. The current prototype only supports three built-in primitive actuators (display a toast message, cause a delay in milliseconds, play a note), but interfaces to the Comingle runtime allow the application developer to implement his/her own domain-specific actuators. The application runtime also include libraries that implement boilerplate routines that help the developer integrate the Comingle rewriting runtime to an `Activity` of the Android SDK.

Network Middleware. As shown in Figure 6, the network middleware provides the underlying communication support between devices running Comingle. We have implemented a concrete instance that utilizes Android's WiFi-direct network protocol to establish connections and send and receive facts between mobile devices. It includes libraries that implement an asynchronous first-in-first-out message sending and receiving service on top of basic network sockets, and libraries that maintains, on each participating location, an active IP address directory of the local ad-hoc network. This allows a group of mobile devices to setup an ad-hoc WiFi-direct network, and supports peer-to-peer communication between any two devices of the group.

5 Case Studies

In this section, we describe three mobile applications we have implemented using the Comingle framework on the Android SDK. Two are multi-player games and one is a networking service. In all three, the overall distributed behavior is orchestrated by the Comingle runtime, while the user interfaces are implemented locally using traditional Android SDK libraries. For brevity, we omit all predicate declarations. These declarations, the code implementing local operations, and the details of the integration with Java and the Android SDK are discussed at length in [12].

Drag Racing. *Drag Racing* is a simple multi-player game inspired by a Google Chrome experiment called Chrome Racer [6]. A number of players compete to reach the finish


```
1  rule init :: [I]initRace(Ls)
2     --o {[A]next(B)|(A,B)<-Cs}, [E]last(),
3         {[I]has(P), [P]all(Ps), [P]at(I), [P]renderTrack(Ls) | P<-Ps}
4         where (Cs,E) = makeChain(I,Ls), Ps = list2mset(Ls).
5
6  rule start :: [X]all(Ps) \ [X]startRace() --o {[P]release()|P<-Ps}.
7
8  rule tap :: [X]at(Y) \ [X]sendTap() --o [Y]recvTap(X).
9
10 rule trans :: [X]next(Z) \ [X]exiting(Y),[Y]at(X) --o [Z]has(Y),[Y]at(Z).
11
12 rule win :: [X]last()\[X]all(Ps),[X]exiting(Y) --o {[P]decWinner(Y)|P<-Ps}.
```

Fig. 7. Drag Racing, a racing game inspired by Chrome Racer

line of a linear racing track. The device of each player shows a distinct segment of the track, and the players advance their car by tapping on their screen. The initial configuration for a three-player instance is shown in Figure 7.[1] In Chrome Racer, the devices interact via a dedicated server. By contrast, the devices in our Drag Racing game communicate with each other directly, without the need of a third party to manage coordination.

An initial configuration such as the one in Figure 7 is generated when rule `init` is executed. Its head is the trigger fact `[I]initRace(Ls)`, where node `I` will hold the initial segment of the track and `Ls` lists all locations participating in the game (including `I`). Several actions need to take place at initialization time, all implemented by the body of `init`. First, the participating locations need to be arranged into a linear chain starting at `I`. This is achieved by the local function *`makeChain`* in the guard `(Cs,E) = makeChain(I,Ls)` where `Cs` is instantiated to a multiset of logically adjacent pairs of locations and `E` to the end of the chain. The guard `Ps = list2mset(Ls)` converts the list `Ls` into a multiset `Ps`. Second, each node other than `E` needs to be informed of which location holds the segment of the track after it, while `E` needs to be told that it has the finishing segment: this is achieved by the atoms `{[A]next(B) | (A,B) <- Cs}` and `[E]last()`, respectively. Third, each location (`P<-Ps`) needs to be informed of who the players are (`[P]all(Ps)`) and of the fact that its car is currently at `I` (`[P]at(I)`), and it needs to be instructed to render the lane of all players (`[P]renderTrack(Ls)`). Fourth, location `I` needs to be instructed to draw the car of all the players (`[I]has(P)`). The facts `renderTrack` and `has` are actuators

[1] In Chrome Racer, the track loops around so that each device shows two segments. While we could easily achieve this effect, our linear "drag" racing variant suffices to demonstrate Comingle's ability to orchestrate distributed computations.

```
1  rule init :: [I]initGame(Ships,Ps)
2       --o [I]turn(), [I]notifyTurn(), {[A]next(B) | (A,B)<-Cs},
3           {[P]all(Ps), [P]randomFleet(Ships) | P <- Ps}
4           where Cs = makeRRchain(Ps).
5
6  rule shoot :: [A]next(B) \ [A]turn(), [A]fireAt(D,X,Y)
7       --o [D]blastAt(A,X,Y), [B]turn(), [B]notifyTurn().
8  rule miss :: [D]empty(X,Y) \ [D]blastAt(A,X,Y)
9       --o [D]missed(A,D,X,Y), [A]missed(A,D,X,Y).
10 rule goodHit :: [D]blastAt(A,X,Y), [D]hull(S,X,Y)
11      --o [D]damaged(S,X,Y), [D]hit(A,D,X,Y), [A]hit(A,D,X,Y).
12 rule dmgHit  :: [D]damaged(S,X,Y) \ [D]blastAt(A,X,Y)
13      --o [D]hit(A,D,X,Y), [A]hit(A,D,X,Y).
14
15 rule sunk :: [D]all(Ps) \
16      [D]damaged(S,X,Y), {[D]damaged(S,X',Y')|(X',Y')->Ds'}
17      {[D]hull(S,W,V)|(W,V)->Hs} | size(Hs)=0
18      --o {[P]sunk(D,S,Ds)|P<-Ps}, [D]checkFleet()
19          where Ds = insert((X,Y), Ds').
20
21 rule deadFleet :: [D]all(Ps), [D]checkFleet(), {[D]checkFleet()},
22      {[D]hull(S,W,V)|(S,W,V)->Hs} | size(Hs)=0
23      --o {[P]notifyDead(D), [P]dead(D) | P<-Ps}.
24
25 rule winner :: [D]all(Ps), {[D]dead(O) | O->Os}
26      | Ps=insert(D,Os) --o {[P]notifyWinner(D) | P<-Ps}.
```

Fig. 8. Multi-way Battleship

since they cause a local computation in the form of screen display. Because the instances of the last four predicate forms are determined by the same multiset (`Ps`), Comingle allows combining them into a single comprehension pattern.

At this point the game has been initialized, but it has not started yet. The race starts the first time a player `X` taps his/her screen. This has the effect of depositing the trigger `[X]startRace()` in the rewriting state, which enables rule `start`. Its body broadcasts the actuator `[P]release()` to every node `P`, which has the effect of informing `P`'s local runtime that subsequent taps will cause its car to move forward. This behavior is achieved by rule `tap`, which is triggered at any node `X` by the fact `[X]sendTap()`, generated by the application runtime every time `X`'s player taps his/her screen. The trigger `[X]exiting(Y)` is generated when the car of player `Y` reaches the right-hand side of the track segment on `X`'s device. If the track continues on player `Z`'s screen (`[X]next(Z)`), rule `trans` hands `Y`'s car over to `Z` by ordering `Z` to draw it on his/her screen (`[Z]has(Y)`) and by informing `X` of the new location of his/her car (`[Y]at(Z)`). Notice that, because fact `[Y]at(X)` is in the simplified head of the rule, it gets consumed. If instead `X` holds the final segment of the track (`[X]last()`) when the trigger `[X]exiting(Y)` materializes, `Y`'s victory is broadcast to all participating locations (`{[P]decWinner(Y) | P <- Ps}`). Besides displaying a banner, it disables moving one's car by tapping the screen.

Multi-way Battleship. Multi-way Battleship extends the classic battleship game with support for more than just two players. Each player begins with an equal assortment of battleships of varying sizes, randomly placed on a two-dimensional grid of cells. The players then take turns selecting an opponent's cell and firing at it. A battleship is sunk when each cell it resides in is hit at least once. The winner of the game is the last player with at least one unsunk ship.

Figure 8 shows a Comingle program that orchestrates this game. Rule `init` initializes an instance of the game. Its head is the trigger `[I]initGame(Ships,Ps)`, where node `I` is the player who will fire the first shot, `Ships` lists the number of ships of each kind, and `Ps` is the multiset of device locations playing the game. Its body informs `I` that it is its turn to play by means of the fact `[I]turn()` and inserts the actuator `[I]notifyTurn()` which posts a notification on `I`'s display and enables touchscreen input. The body of `init` also constructs a round robin sequence of facts `[A]next(B)`, distributes the location of all participants (`[P]all(Ps)`), and deposits the actuator `[P]randomFleet(Ships)` at each node `P`. The application layer of `P` will service this actuator by generating a random placement of the fleet in `Ships` at node `P` and by installing triggers `[P]empty(X,Y)` and `[P]hull(S,X,Y)` to indicate that cell `(X,Y)` is empty or contains a portion of ship `S`, respectively.

The trigger `[A]fireAt(D,X,Y)` is added to the rewriting state when player `A` fires at cell `(X,Y)` of player `D`. It enables rule `shoot`, but only if it is `A`'s turn. This results in the fact `[D]blast(A,X,Y)` added at `D`'s. This rule also passes the turn to the next player (`[A]next(B)`) by asserting the fact `[B]turn()` and causing a notification on `B`'s display (`[B]notifyTurn()`).

The next three rules implement the possible outcomes of such a shot. Specifically, if cell `(X,Y)` is empty, rule `miss` renders an appropriate animation on `A`'s and `B`'s display via the actuator `missed(A,D,X,Y)`. If ship `S` is (partially) in cell `(X,Y)`, rule `goodHit` replaces the fact `[D]hull(S,X,Y)` with `[D]damage(S,X,Y)` and informs `A` and `D` of this event via the actuator `hit(A,D,X,Y)`. If a damaged hull is hit again, rule `dmgHit` generates the `hit(A,D,X,Y)` actuators once more.

Rule `sunk` handles the sinking of a ship `S`. It is enabled if there is at least one fact `[D]damaged(S,X,Y)` in the rewriting state. It then checks that `S` has no intact fragment ({`[D]hull(S,W,V)` | `(W,V)->Hs`} | *size*`(Hs)=0`), collects the coordinates of the other hit fragments ({`[D]damaged(S,X',Y')` | `(X',Y')->Ds'`}), notifies each player that `S` has sunk ({`[P]sunk(D,S,Ds)` | `P<-Ps`}), and issues the fact `[D]checkFleet()` to check if the game is over for `D`. The function *insert* inserts an element in a multiset.

If at least one `[D]checkFleet()` fact is present, rule `deadFleat` similarly checks that no ship fragment is intact ({`[D]hull(S,W,V)` | `(S,W,V)->Hs`} | *size*`(Hs)=0`) and if this is the case it informs all players of `D`'s annihilation with {`[P]notifyDead(D), [P]dead(D)` | `P<-Ps`}. Finally, rule `winner` is executed by the winning player `D` when it can ascertain that all other players are dead (`[D]all(Ps)` {`[D]dead(O)` | `O->Os`} where `Ps=insert(D,Os)`).

WiFi-Direct Directory. WiFi-Direct Directory is an implementation of a networking service built on top of the Android SDK WiFi-direct library. In the WiFi-direct protocol, one device is designated as the *owner* of a newly established group. The owner can

```
1  rule owner  :: [O]startOwner(C)  --o [O]owner(C), [O]joined(O).
2  rule member :: [M]startMember(C) --o [M]member(C).
3  rule connect :: [M]member(C) \ [M]connect(N)
4        --o [O]joinRequest(C,N,M) where O = ownerLoc().
5
6  rule join :: [O]owner(C), {[O]joined(M')|M'->Ms},
7              \ [O]joinRequest(C,N,M) | notIn(M,Ms)
8        --o {[M']added(D)|M'<-Ms}, {[M]added(D')|D'<-Ds},
9            [M]added(D), [O]joined(M), [M]connected()
10           where IP = lookupIP(M), D = (M,IP,N), Ds = retrieveDir().
11
12 rule quitO :: [O]owner(C), [O]quit(), {[O]joined(M)|M->Ms}
13       --o {[M]ownerQuit()|M<-Ms} .
14
15 rule quitM :: {[O]joined(M')|M'->Ms.not(M' = M)}
16               \ [M]member(C), [M]quit(), [O]joined(M)
17       --o {[M']removed(M)|M'<-Ms}, [M]deleteDir().
```

Fig. 9. WiFi-Direct Directory

obtain the IP address of each device in the group from its network middleware, but the other members only know the owner's IP address and location. This means that, initially, the group owner can communicate with all members but the members can only communicate with the owner. WiFi-Direct Directory disseminates and maintains an IP address table on each node of the group in order to enable peer-to-peer IP socket communication.

Figure 9 shows the Comingle program that orchestrates this service. Once the group has been established, the triggers `[O]startOwner(C)` and `[M]startMember(C)` are entered in the rewriting state of the owner and of each other member `M`, respectively. The argument `C` identifies the application this group is for (e.g., one of the two games seen earlier) — the WiFi-direct protocols allows a node to be part of at most one group at any time. Rule `owner` initializes the owner by adding the facts `[O]owner(C)` that sets `O`'s role as the owner of the group for application `C` and `[O]joined(O)` that identifies it as having joined the group. Rule `member` simply sets `M`'s role as a group member (`[M]member(C)`).

The runtime of a member `M` also periodically generates triggers `[M]connect(N)` where `N` is the device's screen name — this is to protect against message losses while the group owner bootstraps. Rule `connect` turn this trigger into the request `[O]joinRequest(C,N,M)` to be sent to the owner `O` — the library function *ownerLoc* retrieves the owner of the current group, which is initially available to all members. This request is processed in rule `join`: the owner `O` checks that a join request by the same member has not been serviced already (`[O]joinRequest(C,N,M) | notIn(M,Ms)`), it then records `M` as having joined the group (`[O]joined(M)`), sends its location, IP address and screen name (`D = (M,IP,N)`) to the active members ({`[M']added(D)|M'<-Ms`}). This same data is sent to `M` (`[M]added(D)`) as well as information about each active member ({`[M]added(D')|D'<-Ds`}). The actuator `[X]added(D)` updates node `X`'s internal routing table with entry `D` and the actuator `[M]connected()` stops the issuance of the triggers `[M]connect(N)`.

The last two rules handle a member `M` leaving the group, which is initiated by trigger `[M]quit()`. If this member is the owner, rule `quitO` dismantles the group and send the actuator `ownerQuit()` to each active member. If `M` is a regular member, rule `quitM` consumes `M`'s `[O]joined(M)` fact, notifies all other members to remove `M`'s entry from their local directory ($\{$`[M']removed(M)|M'<-Ms`$\}$) and instructs `M`'s runtime to delete its entire local directory (`[M]deleteDir()`).

6 Related Work

To the best of our knowledge, Comingle is the first framework to introduce the logic programming paradigm to the development of applications on modern mobile devices. However, it draws from work on distributed and parallel programming languages for decentralized micro-systems, which we now review.

Comingle is greatly influenced by Meld [1], a logic programming language initially designed for programming distributed ensembles of communicating robots. It used the Blinky Blocks platform [8] as a proof of concept to demonstrate simple ensemble programming behaviors. Meld was based on a variant of Datalog extended with sensing and action facts. Recent refinements [2] extended Meld with comprehension patterns and linearity, but refocused it on distributed programming of multicore architectures.

Sifteo [13] is an interactive system that runs an array of puzzle games on Lego-like cubes. Each cube is equipped with a small LCD screen and various means of interaction with the user (e.g., tilting, shaking) and is capable of sensing alignments with neighboring cubes. Developers can implement new games in C/C++ via the Sifteo SDK. Sifteo's decentralized and interactive setup makes it a suitable target platform for Comingle.

The Comingle language is a descendant of CHR [5], a logic programming language targeting traditional constraint solving problems. Comingle extends it with multiset comprehension, explicit locations, triggers and actuators.

7 Future Developments and Conclusions

In this paper, we introduced Comingle, a distributed logic programming language for orchestrating decentralized ensembles. It is designed to simplify the development of interactive applications and to provide a high-level programming abstraction for coordinating distributed computations. As proof of concept, we described three distributed applications orchestrated by Comingle and running on Android mobile devices — two are multi-player games and one is a networking service. By segregating all communication and coordination events in a few rules, it promotes a system-centric, declarative style of programming a distributed application, which simplifies detecting errors and ensuring correctness.

In the immediate future, we intend to expand the language capabilities to capture recurrent synchronization patterns and enrich the programming primitives available at the Comingle level. We will also extend the library support for developing applications that integrate with the Comingle rewriting runtime.

References

1. Ashley-Rollman, M.P., Lee, P., Goldstein, S.C., Pillai, P., Campbell, J.D.: A Language for Large Ensembles of Independently Executing Nodes. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 265–280. Springer, Heidelberg (2009)
2. Cruz, F., Rocha, R., Goldstein, S.C., Pfenning, F.: A linear logic programming language for concurrent programming over graph structures. In: ICLP 2014, Vienna, Austria (2014)
3. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
4. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: OSDI 2004. USENIX Association (2004)
5. Frühwirth, T., Raiser, F.: Constraint Handling Rules: Compilation, Execution and Analysis (2011), BOD ISBN 9783839115916
6. New York Google. Chrome Racer, A Chrome Experiment (2013), http://www.chrome.com/racer
7. Google Inc. Google Web Toolkit, http://code.google.com/webtoolkit/
8. Kirby, B.T., Ashley-Rollman, M., Goldstein, S.C.: Blinky blocks: A physical ensemble programming platform. In: CHI 2011, pp. 1111–1116. ACM, New York (2011)
9. Lam, E.S.L., Cervesato, I.: Decentralized Execution of Constraint Handling Rules for Ensembles. In: PPDP 2013, Madrid, Spain, pp. 205–216 (2013)
10. Lam, E.S.L., Cervesato, I.: Optimized Compilation of Multiset Rewriting with Comprehensions. In: Garrigue, J. (ed.) APLAS 2014. LNCS, vol. 8858, pp. 19–38. Springer, Heidelberg (2014)
11. Lam, E.S.L., Cervesato, I.: Reasoning about Set Comprehension. In: SMT 2014 (2014)
12. Lam, E.S.L., Cervesato, I.: Comingle: Distributed Logic Programming for Decentralized Android Applications. Technical Report CMU-CS-15-101, Carnegie Mellon University (March 2015)
13. Merrill, D., Kalanithi, J.: Sifteo, Interactive Game Cubes (2009), https://www.sifteo.com/cubes cubes
14. Stuckey, P.J., Sulzmann, M., Wazny, J.: Interactive Type Debugging in Haskell. In: Haskell 2003, pp. 72–83. ACM, New York (2003)