

# Comingle: Distributed Logic Programming for Decentralized Android Applications

**Edmund S. L. Lam and Iliano Cervesato**

March 2015  
CMU-CS-15-101  
CMU-CS-QTR-125

School of Computer Science  
Carnegie Mellon University  
Pittsburgh, PA 15213

Carnegie Mellon University, Qatar campus.  
The author can be reached at [sllam@qatar.cmu.edu](mailto:sllam@qatar.cmu.edu) or [iliano@cmu.edu](mailto: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 three distributed mobile games and one networking service orchestrated using Comingle.

\* This paper 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.

**Keywords:** Multiset Rewriting, Logic Programming, Android Mobile Programming

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>A Motivating Example</b>	<b>1</b>
<b>3</b>	<b>Abstract Syntax and Semantics</b>	<b>3</b>
3.1	Abstract Syntax . . . . .	3
3.2	Abstract Semantics . . . . .	4
<b>4</b>	<b>Implementation</b>	<b>6</b>
4.1	Compilation . . . . .	6
4.2	Runtime System . . . . .	7
<b>5</b>	<b>Case Studies</b>	<b>8</b>
5.1	Drag Racing . . . . .	8
5.2	Multiway Battleship . . . . .	15
5.3	WiFi-Direct Directory . . . . .	16
5.4	Swarbble . . . . .	19
<b>6</b>	<b>Related Work</b>	<b>22</b>
<b>7</b>	<b>Future Developments and Conclusions</b>	<b>22</b>
<b>A</b>	<b>Compiled Drag Racing Example</b>	<b>23</b>

## List of Figures

1	Pivot Swap, orchestrated by Comingle . . . . .	2
2	Abstract Syntax and Runtime Artifacts of Comingle . . . . .	3
3	Matching a Rule Head . . . . .	4
4	Processing a Rule Body . . . . .	5
5	Abstract Semantics of Comingle . . . . .	6
6	Runtime System of a Distributed Comingle Mobile Application . . . . .	7
7	Drag Racing, a Racing Game Inspired by Chrome Racer . . . . .	9
8	Creating an instance of the Drag Racing Runtime in an Android Activity . . . . .	11
9	Associating Comingle Actuators to the Drag Racing Android Activity . . . . .	12
10	Triggering Comingle Multiset Rewriting . . . . .	13
11	Housekeeping Code for the Comingle Runtime . . . . .	14
12	Battleship, Mid-play Screen Shot . . . . .	15
13	Multiway Battleship (Declarations) . . . . .	16
14	Multiway Battleship (Rules) . . . . .	17
15	WiFi-Direct Directory . . . . .	18
16	Swarbble, Mid-play Screen Shot . . . . .	19
17	Swarbble (Declarations) . . . . .	20
18	Swarbble (Rules) . . . . .	21
19	Drag Racing Compilation: Facts, Triggers and Actuators . . . . .	24
20	Drag Racing Compilation: Rule Compilation Example . . . . .	25

# 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 node-centric style mentioned above. This system-centric view underlies popular frameworks such as Google Web Toolkit [2] (for client-server web development) and Map Reduce [5] (for parallel distributed algorithms on large-scale computing clusters). In earlier work [9, 11], 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, 11]. 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 [11]. 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.

## 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*,

```

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 }.

```

Figure 1: Pivot Swap, orchestrated by Comingle

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  $[l]p(\vec{t})$  where  $p$  is 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  $[l]swap(l', P)$  represents  $l$ 's intent to swap data with device  $l'$  based on the pivot value  $P$ , fact  $[l]item(I)$  indicates that value  $I$  is held at location  $l$ , while  $[l]display(S)$  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  $\{[X]item(I) \mid I \rightarrow Is. I \leq P\}$  identifies all of  $X$ 's items  $I$  such that  $I \leq P$ . Similarly, all of  $Y$ 's items  $J$  such that  $J \geq P$  are identified by  $\{[Y]item(J) \mid J \rightarrow Js. J \geq P\}$ . 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  $item(I)$  are meaningful only at the rewriting level. Facts are also used as an interface to a device's local computations. Specifically, facts like  $[l]swap(l', P)$  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  $[l]swap(l', P)$  for emphasis, are only allowed in the heads of a rule. Dually, facts like  $[l]display(S)$  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  $[l]\underline{display}(S)$ , 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 of the form  $[l]\underline{display}(S)$  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  $[l]\underline{display}(S)$ .

By being system-centric, the code in Figure 1 lets the developer think in terms of overall behavior rather than

Locations: $l$	Terms: $t$	Guards: $g$	Standard / trigger / actuator predicates: $p_s, p_t, p_a$
Standard facts $F_s ::= [l] p_s(\vec{t})$	Triggers $F_t ::= [l] p_t(\vec{t})$	Actuators $F_a ::= [l] p_a(\vec{t})$	
Facts $f, F ::= F_s \mid F_t \mid F_a$			
Head atoms $h ::= F_s \mid F_t$	Body atoms $b ::= F_s \mid F_a$		
Head expressions $H ::= h \mid \lambda h \mid g \int_{\vec{x} \in t}$	Body expressions $B ::= b \mid \lambda b \mid g \int_{\vec{x} \in t}$		
Comingle rule $R ::= \overline{H} \setminus \overline{H} \mid g \multimap \overline{B}$	Comingle program $\mathcal{P} ::= \overline{R}$		
Local state: $[l]\psi$			
Rewriting state $St ::= \overline{F}$	Application state $\Psi ::= \overline{[l]\psi}$	Comingle state $\Theta ::= \langle St; \Psi \rangle$	

Figure 2: Abstract Syntax and Runtime Artifacts of Comingle

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  $\bar{o}$  for a multiset of syntactic objects  $o$ . We denote the extension of a multiset  $\bar{o}$  with an object  $o$  as “ $\bar{o}, o$ ”, with  $\emptyset$  indicating the empty multiset. We also write “ $\bar{o}_1, \bar{o}_2$ ” for the union of multisets  $\bar{o}_1$  and  $\bar{o}_2$ . 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  $\alpha$ -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  $[l] p(\vec{t})$ . 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 *propagated* and the *simplified 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  $\lambda f \mid g \int_{\vec{x} \in t}$ . An atom  $f$  is a located fact  $[l] p(\vec{t})$  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  $\lambda f \mid g \int_{\vec{x} \in t}$  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

Matching:  $\overline{H} \triangleq_{\text{lhs}} St \quad H \triangleq_{\text{lhs}} St$

$$\frac{\overline{H} \triangleq_{\text{lhs}} St \quad H \triangleq_{\text{lhs}} St'}{\overline{H}, H \triangleq_{\text{lhs}} St, St'}^{(l_{\text{mset-1}})} \quad \frac{}{\emptyset \triangleq_{\text{lhs}} \emptyset}^{(l_{\text{mset-2}})} \quad \frac{}{F \triangleq_{\text{lhs}} F}^{(l_{\text{fact}})}$$

$$\frac{[\vec{t}/\vec{x}]f \triangleq_{\text{lhs}} F \quad \models [\vec{t}/\vec{x}]g \quad \lambda f \mid g \int_{\vec{x} \in \vec{ts}} \triangleq_{\text{lhs}} St}{\lambda f \mid g \int_{\vec{x} \in \vec{t}, \vec{ts}} \triangleq_{\text{lhs}} St, F}^{(l_{\text{comp-1}})} \quad \frac{}{\lambda f \mid g \int_{\vec{x} \in \emptyset} \triangleq_{\text{lhs}} \emptyset}^{(l_{\text{comp-2}})}$$

Residual Non-matching:  $\overline{H} \triangleq_{\text{lhs}}^\neg St \quad H \triangleq_{\text{lhs}}^\neg St$

$$\frac{\overline{H} \triangleq_{\text{lhs}}^\neg St \quad H \triangleq_{\text{lhs}}^\neg St}{\overline{H}, H \triangleq_{\text{lhs}}^\neg St}^{(l_{\text{mset-1}}^\neg)} \quad \frac{}{\emptyset \triangleq_{\text{lhs}}^\neg St}^{(l_{\text{mset-2}}^\neg)} \quad \frac{}{F \triangleq_{\text{lhs}}^\neg St}^{(l_{\text{fact}}^\neg)}$$

$$\frac{F \not\triangleq_{\text{lhs}} \lambda f \mid g \int_{\vec{x} \in ts} \quad \lambda f \mid g \int_{\vec{x} \in ts} \triangleq_{\text{lhs}}^\neg St}{\lambda f \mid g \int_{\vec{x} \in ts} \triangleq_{\text{lhs}}^\neg St, F}^{(l_{\text{comp-1}}^\neg)} \quad \frac{}{\lambda f \mid g \int_{\vec{x} \in ts} \triangleq_{\text{lhs}}^\neg \emptyset}^{(l_{\text{comp-2}}^\neg)}$$

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

Figure 3: Matching a Rule Head

atom  $f$  and the guard  $g$ . We implicitly  $\alpha$ -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  $\lambda f \mid g \int_{\vec{x} \in t}$  as  $\{f \mid \vec{x} \rightarrow t. g\}$  in rule heads and  $\{f \mid \vec{x} \leftarrow t. g\}$  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  $\mathcal{P} \triangleright \Theta \mapsto \Theta'$  where  $\mathcal{P}$  is a program,  $\Theta$  is a *state* and  $\Theta'$  is a state that can be reached in one (abstract) step of computation. A state  $\Theta$  has the form  $\langle St; \Psi \rangle$ . The first component  $St$  is a collection of ground located facts  $[l] p(\vec{t})$  and is called the *rewriting state* of the system. Comingle rules operate exclusively on the rewriting state. The second component, the *application state*  $\Psi$ , is the collection of the *local states*  $[l]\psi$  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  $\Psi$  but can also consume triggers from the rewriting state and add actuators into it. These runtime artifacts are formally defined at the bottom of Figure 2.

We will now describe the two types of state transitions  $\mathcal{P} \triangleright \Theta \mapsto \Theta'$  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  $\overline{H}_p \setminus \overline{H}_s \mid g \rightarrow \overline{B}$  involves two main operations: identifying fragments of the rewriting state  $St$  that match the rule heads  $\overline{H}_p$  and  $\overline{H}_s$ , and extending 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



Unfolding Rule Body:  $\overline{B} \gg_{\text{rhs}} St \quad B \gg_{\text{rhs}} St$

$$\begin{array}{c}
 \frac{\overline{B} \gg_{\text{rhs}} St \quad B \gg_{\text{rhs}} St'}{\overline{B}, B \gg_{\text{rhs}} St, St'} (r_{mset-1}) \quad \frac{}{\emptyset \gg_{\text{rhs}} \emptyset} (r_{mset-2}) \quad \frac{}{F \gg_{\text{rhs}} F} (r_{fact}) \\
 \frac{\models [\vec{t}/\vec{x}]g \quad [t/\vec{x}]b \gg_{\text{rhs}} F \quad \lambda b \mid g \int_{\vec{x} \in \vec{ts}} \gg_{\text{rhs}} St}{\lambda b \mid g \int_{\vec{x} \in \vec{t}, ts} \gg_{\text{rhs}} F, St} (r_{comp-1}) \\
 \frac{\not\models [\vec{t}/\vec{x}]g \quad \lambda b \mid g \int_{\vec{x} \in \vec{ts}} \gg_{\text{rhs}} St}{\lambda b \mid g \int_{\vec{x} \in \vec{t}, ts} \gg_{\text{rhs}} St} (r_{comp-2}) \quad \frac{}{\lambda b \mid g \int_{\vec{x} \in \emptyset} \gg_{\text{rhs}} \emptyset} (r_{comp-3})
 \end{array}$$

Figure 4: Processing a Rule Body

account). Further details can be found in [10].

**Matching Rule Heads** Let  $\overline{H}$  be a (propagated or simplified) 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  $\overline{H} \triangleq_{\text{lhs}} St^+$  defined in the top part of Figure 3. Rules  $\downarrow_{mset-*}$  partition  $St^+$  into fragments to be matched by each atom in  $\overline{H}$ : plain facts  $F$  must occur identically (rule  $\downarrow_{fact}$ ) while for comprehension atoms  $\lambda f \mid g \int_{\vec{x} \in \vec{ts}}$  the state fragment must contain a distinct instance of  $f$  for every element of the comprehension range  $\vec{ts}$  that satisfies the comprehension guard  $g$  (rules  $\downarrow_{comp-*}$ ).

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  $\overline{H} \triangleq_{\text{lhs}}^- St^-$  in the bottom part of Figure 3. Rules  $\downarrow_{mset-*}^-$  tests each individual atom and rule  $\downarrow_{fact}^-$  ignore facts. Rules  $\downarrow_{comp-*}^-$  deal with comprehensions  $\lambda f \mid g \int_{\vec{x} \in \vec{ts}}$ : they check that no fact in  $St^-$  matches any instance of  $f$  while satisfying  $g$  — note that the comprehension range  $\vec{ts}$  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  $r_{mset-*}$  go through  $\overline{B}$ . Atomic facts  $F$  are added immediately (rule  $r_{fact}$ ). Instead, comprehension atoms  $\lambda f \mid g \int_{\vec{x} \in \vec{ts}}$  need to be *unfolded* (rules  $r_{comp-*}$ ): for every item  $\vec{t}$  in  $\vec{ts}$  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  $\theta$ . 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 propagated and simplified heads must match fragments  $St_p$  and  $St_s$  respectively ( $\theta \overline{H}_p \triangleq_{\text{lhs}} St_p$  and  $\theta \overline{H}_s \triangleq_{\text{lhs}} St_s$ ), while the remaining fragment  $St$  must be free of residual matchings ( $\theta(\overline{H}_p, \overline{H}_s) \triangleq_{\text{lhs}}^- St$ ). The rule body instance  $\theta \overline{B}$  is then unfolded ( $\theta \overline{B} \gg_{\text{rhs}} St_b$ ) 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. A description of this process is beyond the scope of the present paper but can be found in [9].

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

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

$$\frac{\frac{\frac{\theta \overline{H}_p \triangleq_{\text{lhs}} St_p \quad \theta \overline{H}_s \triangleq_{\text{lhs}} St_s \quad \theta(\overline{H}_p, \overline{H}_s) \triangleq_{\text{lhs}} St \quad \theta \overline{B} \ggg_{\text{rhs}} St_b}{\mathcal{P} \triangleright \langle St_p, St_s, St; \Psi \rangle \mapsto \langle St_p, St_b, St; \Psi \rangle} \quad \models \theta g}{\mathcal{P} \triangleright \langle St, [l]\mathcal{A}; \Psi, [l]\psi \rangle \mapsto \langle St, [l]\mathcal{T}; \Psi, [l]\psi' \rangle} \quad (\text{rw.ens})}{\frac{\langle \mathcal{A}; \psi \rangle \mapsto_l \langle \mathcal{T}; \psi' \rangle}{\mathcal{P} \triangleright \langle St, [l]\mathcal{A}; \Psi, [l]\psi \rangle \mapsto \langle St, [l]\mathcal{T}; \Psi, [l]\psi' \rangle} \quad (\text{rw.loc})}$$

Figure 5: Abstract Semantics of Comingle

**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  $\psi$  into  $\psi'$ , 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  $[l]\mathcal{A}$  for a portion of the actuators located at  $l$  in the current rewriting state — there may be others. We similarly write  $[l]\mathcal{T}$  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. 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 pre-processor 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 [15] 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 [4]. Our SMT solver library includes an extension to reason about set comprehensions [13] 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 [11]. 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 compilation scheme formalized in [11] that first

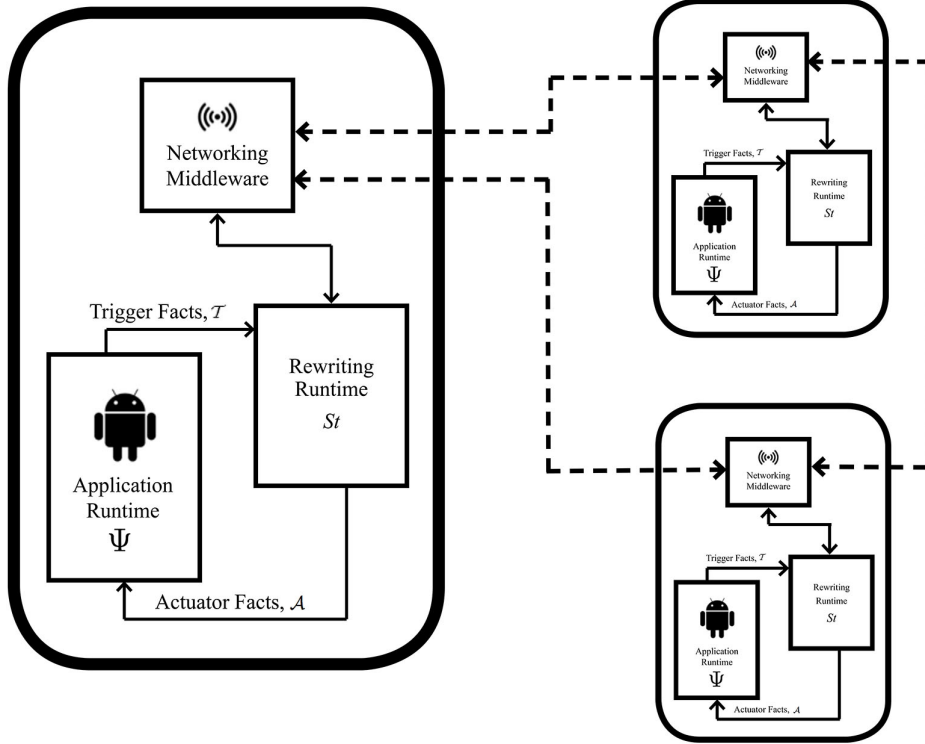


Figure 6: Runtime System of a Distributed Comingle Mobile Application

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. Details of this compilation scheme can be found in [11].

## 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. This operational semantics also exploits the committed-choice nature of the abstract semantics and selects an optimal execution orderings. 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 four mobile applications we have implemented using the Comingle framework on the Android SDK. Three of them are multi-player games whose game logic and distributed behavior are orchestrated by the Comingle runtime, while the user interfaces are implemented locally using traditional Android SDK libraries. A fourth is a lower-level networking service that we provides support to other Comingle applications. For the first example (Drag Racing), we show the Comingle program that orchestrates the distributed computations of the application, as well as the details of how the Drag Racing Comingle runtime is integrated into the Android application that implements local computations.

### 5.1 Drag Racing

*Drag Racing* is a simple multi-player game inspired by a Google Chrome experiment called Chrome Racer [7]. A number of players compete to reach the finish 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.<sup>1</sup> 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.

---

<sup>1</sup>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  module comingle.lib.ExtLib import {
2    mset :: [A] -> A.
3  }
4
5  module comingle.dragracing.RacerLib import {
6    makeChain :: (A, [A]) -> ({(A,A)}, A).
7  }
8
9  predicate initRace      :: [loc] -> trigger.
10 predicate startRace    :: trigger.
11 predicate sendTap     :: trigger.
12 predicate exiting    :: loc -> trigger.
13
14 predicate at          :: loc -> fact.
15 predicate rmAt       :: loc -> fact.
16 predicate next       :: loc -> fact.
17 predicate last       :: fact.
18 predicate all        :: loc -> fact.
19
20 predicate renderTrack :: [loc] -> actuator.
21 predicate release    :: actuator.
22 predicate recvTap    :: loc -> actuator.
23 predicate has        :: loc -> actuator.
24 predicate decWinner  :: loc -> actuator.
25
26 rule init :: [I]initRace(Ls)
27   --o {[A]next(B) | (A,B) <- Cs}, [E]last(),
28   {[I]has(P), [P]all(Ps), [P]at(I), [P]renderTrack(Ls) | P <- Ps}
29   where (Cs,E) = makeChain(I,Ls), Ps = list2mset(Ls).
30
31 rule start :: [X]all(Ps) \ [X]startRace() --o {[P]release() | P <- Ps}.
32
33 rule tap   :: [X]at(Y) \ [X]sendTap() --o [Y]recvTap(X).
34
35 rule trans :: [X]next(Z) \ [X]exiting(Y), [Y]at(X)
36   --o [Z]has(Y), [Y]at(Z).
37
38 rule win  :: [X]last() \ [X]all(Ps), [X]exiting(Y)
39   --o {[P]decWinner(Y) | P <- Ps}.

```

Figure 7: Drag Racing, a Racing Game Inspired by Chrome Racer

**The Comingle Program** 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  $\{[A]next(B) \mid (A, B) \leftarrow Cs\}$  and `[E]last()`, respectively. Third, each location  $(P \leftarrow Ps)$  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 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 ( $\{[P]decWinner(Y) \mid P \leftarrow Ps\}$ ). Besides displaying a banner, it disables moving one's car by tapping the screen.

**Integration with Java and Android SDK** We now show the details of how the the Drag racing Comingle code in Figure 7 is integrated into the actual Android application, implemented in Java and the Android SDK library, that runs on a user's mobile device. The full code for this Android project (including the other examples shown in this section) are available on the Comingle Github repository at [https://github.com/sllam/comingle/tree/master/android\\_apps](https://github.com/sllam/comingle/tree/master/android_apps)

Each Comingle program is compiled into a Java class that extends `RewriteMachine` of the Comingle runtime library. This super class provides the basic functionality and general interfaces that allows Comingle programs to be executed as background daemon routines (e.g., `start` and `stop` administrative operations, generic query interfaces to retrieve intermediate states of the fact stores). Each generated instance of this class concretely implement multiset matching routines, internally executed by the Comingle runtime. For the Drag Racing program, the compiler generates `Dragracing.java` containing the class `Dragracing` that implements the multiset rewriting specified by the Drag Racing Comingle program. The generated class is independent from the Android SDK library, hence can be easily adapted to execute on any machine equipped with a Java virtual machine. Excerpts of the resulting code are displayed in Appendix A.

Our Comingle runtime library includes APIs that allows developers to implement such simulated executions, to facilitate unit testing of Comingle programs. To allow easy integration with the Android SDK library, our runtime library also includes APIs that specializes Comingle runtime instances for decentralized execution on Android devices. The current implementation provides this support in the form of a wrapper class `WifiDirectComingleRuntime`, that implements decentralized execution of Comingle programs over the WiFi-Direct network. As shown in Figure 8, we first create an instance of the class `WifiDirectComingleRuntime` in the activity's `onCreate` method (lines 6-7). As inputs, the developer provides (1) a reference to the host Android activity (`this`) in which the Comingle runtime is running on, (2) the subclass of the `RewriteMachine` to create (`Dragracing.class`), (3) a string identifier of the Drag Racing application ("`comingle.example.dragracing`") and (4) two port numbers for administra-

```

1  @Override
2  public void onCreate(Bundle savedInstanceState) {
3
4      ... // Other non-Comingle related initialization operations
5
6      dragracingRuntime =
7          new WifiDirectComingleRuntime<Dragracing>
8              (this, Dragracing.class, "comingle.example.dragracing",
9                DR_ADMIN_PORT, DR_FACT_PORT);
10     dragracingRuntime.initWifiDirectEnvironment();
11
12     dragracingRuntime.addLocalNodeInfoAvailableListener
13         (new LocalNodeInfoAvailableListener() {
14             public void doLocalNodeInfoAvailableAction
15                 (NodeInfo local, final int role) {
16                 startDragRacingRewriteMachine();
17             }
18         });
19 }

```

Figure 8: Creating an instance of the Drag Racing Runtime in an Android Activity

tive (`DR_ADMIN_PORT`) and fact data (`DR_FACT_PORT`) messages. Network sockets with these port numbers will be opened during the execution of this Comingle runtime. The call to `initWifiDirectEnvironment()` at line 8 initializes the Comingle network middleware by subscribing to channels that listen on WiFi-Direct connection events (via `BroadcastReceiver` of the Android SDK) and by opening network sockets that convey administrative messages between participating devices. This results in a background routine that maintains an active directory service (as shown in Figure 15) for each participating device that runs this application, facilitating peer-to-peer IP socket communication between any participating device. Our library allows the programmer to choose between two implementations of this directory service: one implemented purely in Java, while the other in Java and orchestrated by Comingle. See Section 5.3 for a brief comparison. Lines 10–14 adds a listener routine onto the Comingle runtime that is triggered when the mobile device has successfully connected to a WiFi-Direct group running the Drag Racing application.

This listener runs the operation `startDragRacingRewriteMachine()` defined in Figure 9 that initializes the Drag Racing Comingle runtime. First (line 2) the call to `initRewriteMachine()` sets up the communication sockets of the Comingle runtime. The next sequence of operations (lines 4–27) define the actuators of the Drag Racing Comingle runtime by linking each to a concrete method of the Drag Racing Android activity. For instance, actuator `renderTrack` is associated with a concrete Java method of the same name in lines 4–7, and this Java method (code omitted) draws the track onto the an active canvas on the main view of the Android activity. Similarly, each of the four other actuators of the Drag Racing runtime are mapped to Java methods that implement their respective side-effects. The code for these methods is omitted as it implements routine visualization functionalities that are not central to this report (e.g., rendering sprites on a canvas and other front-end user interface routines). Finally, the Drag Racing Comingle runtime is started when the method `startRewriteMachine()` (line 29) is called. After `startDragRacingRewriteMachine()` has successfully executed, the Comingle runtime instance is ready to orchestrate the Drag Racing game with participating devices.

Figure 10 show the methods of the Drag Racing Android activity that calls the four triggers of the Drag Racing Comingle program. Lines 1–8 implements the `onTouch` operation embedded in the touch listener of the main View of the activity. This operation simply sends a trigger fact `sendTap()` to the Comingle runtime by calling the method `addSendTap()` of the Comingle runtime (line 4). Lines 10–24 implements a standard Android application action bar options menu, which contains two buttons that are initially invisible: an ‘init’ button referenced by `R.id.action.init` that adds the trigger fact `initRace` (line 15), and a ‘go’ button referenced by

```

1 private void startDragRacingRewriteMachine() {
2     dragracingRuntime.initRewriteMachine();
3
4     ActuatorAction<LinkedList<Integer>> renderTrackAction =
5         new ActuatorAction<LinkedList<Integer>>() {
6             public void doAction(LinkedList<Integer> locs) {
7                 renderTrack(locs);
8             }
9         };
10    dragracingRuntime.getRewriteMachine()
11        .setRenderTrackActuator(renderTrackAction);
12
13    ActuatorAction<Unit> releaseAction =
14        new ActuatorAction<Unit>() {
15            public void doAction(Unit arg0) { release(); }
16        };
17    dragracingRuntime.getRewriteMachine()
18        .setReleaseActuator(releaseAction);
19
20    ActuatorAction<Integer> recvTapAction =
21        new ActuatorAction<Integer>() {
22            public void doAction(Integer player_idx) {
23                tap(player_idx);
24            }
25        };
26    dragracingRuntime.getRewriteMachine()
27        .setRecvTapActuator(recvTapAction);
28
29    ActuatorAction<Integer> hasAction =
30        new ActuatorAction<Integer>() {
31            public void doAction(Integer player_idx) {
32                addPlayer(player_idx);
33            }
34        };
35    dragracingRuntime.getRewriteMachine().setHasActuator(hasAction);
36
37    ActuatorAction<Integer> decWinnerAction =
38        new ActuatorAction<Integer>() {
39            public void doAction(Integer winner) {
40                declareWinner(winner);
41            }
42        };
43    dragracingRuntime.getRewriteMachine()
44        .setDecWinnerActuator(decWinnerAction);
45
46    dragracingRuntime.startRewriteMachine();
47 }

```

Figure 9: Associating Comingle Actuators to the Drag Racing Android Activity



```

1 public boolean onTouch(View v, MotionEvent event) {
2     switch(event.getAction()) {
3         case MotionEvent.ACTION_DOWN:
4             if (!released()) {
5                 dragracingRuntime.getRewriteMachine().addSendTap();
6             }
7             return true;
8     }
9     return true;
10 }
11
12 public boolean onOptionsItemSelected(MenuItem item) {
13     int id = item.getItemId();
14     switch (id) {
15         case R.id.action_init:
16             LinkedList<Integer> locs = dragracingRuntime.getDirectory()
17                 .getLocations();
18             dragracingRuntime.getRewriteMachine().addInitRace(locs);
19             setMenuItemVisibility(R.id.action_go, true);
20             setMenuItemVisibility(R.id.action_start, false);
21             return true;
22         case R.id.action_go:
23             dragracingRuntime.getRewriteMachine().addStartRace();
24             return true;
25     }
26     return super.onOptionsItemSelected(item);
27 }
28
29 public void moveCar(Car car, int time_prev, int time_curr) {
30     Paint bg_paint = new Paint();
31     bg_paint.setColor(BG_COLOR);
32     track_canvas.drawRect(car.getSprite(), bg_paint);
33     int new_pos = RacerLib.newPos(car.getSprite().left,
34         time_prev, time_curr);
35     if (new_pos < TRACK_LENGTH) {
36         car.move(new_pos);
37         track_canvas.drawRect(car.getSprite(), car.getPaint());
38     } else {
39         removeCar(car.getOwner());
40         dragracingRuntime.getRewriteMachine()
41             .addExiting(car.getOwner());
42     }
43 }

```

Figure 10: Triggering Comingle Multiset Rewriting

```

1  protected void onResume() {
2      ... // Other non-Comingle related onResume operations
3      dragracingRuntime.registerReceiver();
4  }
5
6  protected void onPause() {
7      ... // Other non-Comingle related onPause operations
8      dragracingRuntime.unregisterReceiver();
9  }
10
11 protected void onDestroy() {
12     ... // Other non-Comingle related onDestroy operations
13     dragracingRuntime.close();
14 }
15
16 protected void setupNotifications() {
17     dragracingRuntime.addDirectoryChangeListener(
18         new DirectoryChangeListener() {
19             public void doDirectoryChangedAction
20                 (final List<NodeInfo> new_peers, List<NodeInfo> added_nodes,
21                  final List<NodeInfo> dropped_nodes, int role) {
22                 if(dropped_nodes.size() > 0) {
23                     postAlert("Player Dropped", "A player has dropped out!");
24                 }
25             }
26         });
27 }

```

Figure 11: Housekeeping Code for the Comingle Runtime

`R.id.action_go` that adds the trigger fact `startRace` (line 20). The ‘go’ button is only made visible after the ‘init’ button is clicked. The ‘init’ button is also initialized so that it is visible to the group owner only. Finally, on lines 26–38 we show the main sprite animation method of the activity, which adds a trigger fact `exiting` (line 36) to the Comingle runtime, when a particular car has reached the end of a track.

Figure 11 shows the remaining fragments of code in the Drag Racing application that references the Comingle runtime. The call to `registerReceiver()` (line 3) within the `onResume` method of the Android activity simply registers the `WifiDirectComingleRuntime` instance to the Android event notification framework. This allows the Comingle runtime to listen and watch for events like addition of a new device in the WiFi-Direct group. The call to `unRegisterReceiver()` (line 8) unregisters the Comingle runtime when the Android application is inactive. The call to `close()` (line 13) closes all network sockets of the Comingle runtime, in preparation for the termination of the Android application. Finally, the method `setupNotifications()` adds a listener that subscribes to the event where the Comingle peer-to-peer directory has changed. This listener posts an alert message to notify the user in the event of a player dropping out from the game.

This concludes all that an Android developer needs to do to integrate the Comingle runtime to the Android application: the problem of orchestrating and coordinating mobile devices to run the Drag Racing game is entirely specified in the Comingle program (Figure 7). In fact, the only remaining Java code of the Android application are (1) the omitted code containing local computations of the actuators shown in Figure 9 and (2) a library function `RacerLib.newPos` that computes new positions of sprites used at line 30 of Figure 10. This local computation code constitute the bulk of the Java code that implements the Android application.

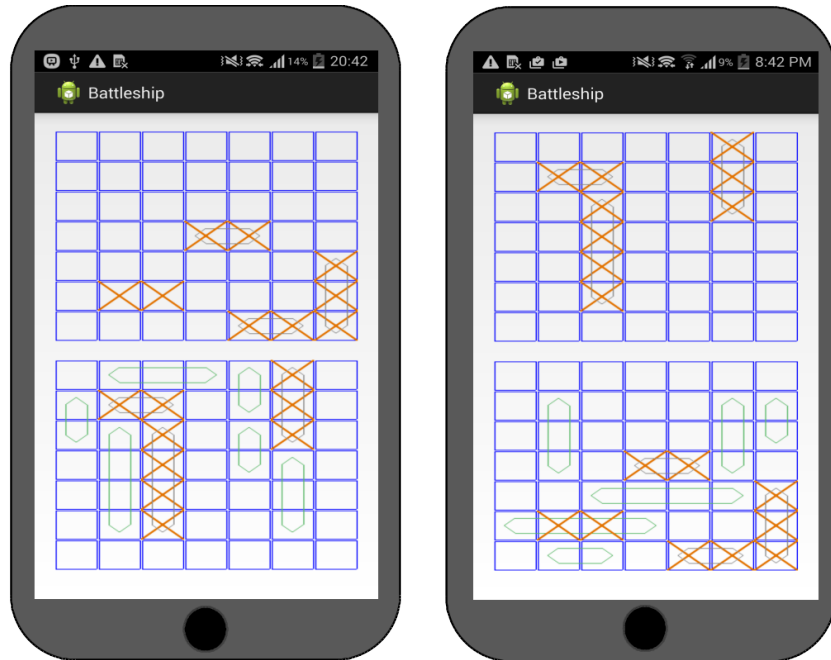


Figure 12: Battleship, Mid-play Screen Shot

## 5.2 Multiway Battleship

Multiway 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 12 shows the screens of a two-player instance.

Figures 13 and 14 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

```

1  module comingle.lib.ExtLib import {
2    union :: ({A},{A}) -> {A}.
3    size  :: {A} -> int.
4    makeRRChain :: {A} -> {(A,A)}.
5  }
6
7  predicate initGame      :: ((int,int,int),[loc]) -> trigger.
8  predicate fireAt       :: (loc,int,int) -> trigger.
9  predicate empty        :: (int,int) -> trigger.
10 predicate hull         :: (string,int,int) -> trigger.
11
12 predicate blastAt      :: (loc,int,int) -> fact.
13 predicate next        :: loc -> fact.
14 predicate turn        :: fact.
15 predicate all         :: {loc} -> fact.
16 predicate damaged    :: (string,int,int) -> fact.
17 predicate checkFleet :: fact.
18 predicate dead        :: loc -> fact.
19
20 predicate randomFleet :: (int,int,int) -> actuator.
21 predicate notifyTurn  :: actuator.
22 predicate missed     :: (loc,loc,int,int) -> actuator.
23 predicate hit        :: (loc,loc,int,int) -> actuator.
24 predicate sunk       :: (loc,loc,string,{(int,int)}) -> actuator.
25 predicate notifyDead  :: loc -> actuator.
26 predicate notifyWinner :: loc -> actuator.

```

Figure 13: Multiway Battleship (Declarations)

rewriting state. It then checks that  $S$  has no intact fragment ( $\{[D] \text{hull}(S, W, V) \mid (W, V) \rightarrow Hs\} \mid \text{size}(Hs)=0$ ), collects the coordinates of the other hit fragments ( $\{[D] \text{damaged}(S, X', Y') \mid (X', Y') \rightarrow Ds'\}$ ), notifies each player that  $S$  has sunk ( $\{[P] \text{sunk}(D, S, Ds) \mid P \leftarrow Ps\}$ ), and issues the fact  $[D] \text{checkFleet}()$  to check if the game is over for  $D$ . The function *insert* inserts an element in a multiset.

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

### 5.3 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 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 15 shows the Comingle program that orchestrates this service. Once the group has been established, the triggers  $[O] \text{startOwner}(C)$  and  $[M] \text{startMember}(C)$  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

```

27 rule init :: [I]initGame(Ships,Ps)
28     --o [I]turn(), [I]notifyTurn(), {[A]next(B) | (A,B)<-Cs},
29     {[P]all(Ps), [P]randomFleet(Ships) | P <- Ps}
30     where Cs = makeRRchain(Ps).
31
32 rule shoot :: [A]next(B) \ [A]turn(), [A]fireAt(D,X,Y)
33     --o [D]blastAt(A,X,Y), [B]turn(), [B]notifyTurn().
34
35 rule miss :: [D]empty(X,Y) \ [D]blastAt(A,X,Y)
36     --o [D]missed(A,D,X,Y), [A]missed(A,D,X,Y).
37
38 rule goodHit :: [D]blastAt(A,X,Y), [D]hull(S,X,Y)
39     --o [D]damaged(S,X,Y), [D]hit(A,D,X,Y), [A]hit(A,D,X,Y).
40
41 rule dmgHit :: [D]damaged(S,X,Y) \ [D]blastAt(A,X,Y)
42     --o [D]hit(A,D,X,Y), [A]hit(A,D,X,Y).
43
44 rule sunk :: [D]all(Ps) \
45     [D]damaged(S,X,Y), {[D]damaged(S,X',Y') | (X',Y')->Ds'}
46     {[D]hull(S,W,V) | (W,V)->Hs} | size(Hs)=0
47     --o {[P]sunk(D,S,Ds) | P<-Ps}, [D]checkFleet()
48     where Ds = insert((X,Y), Ds').
49
50 rule deadFleet :: [D]all(Ps), [D]checkFleet(), {[D]checkFleet()},
51     {[D]hull(S,W,V) | (S,W,V)->Hs} | size(Hs)=0
52     --o {[P]notifyDead(D), [P]dead(D) | P<-Ps}.
53
54 rule winner :: [D]all(Ps), {[D]dead(O) | O->Os} | Ps=insert(D,Os)
55     --o {[P]notifyWinner(D) | P<-Ps}.

```

Figure 14: Multiway Battleship (Rules)

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 information 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

```

1  module comingle.lib.ExtLib import {
2      not :: bool -> bool
3  }
4
5  module p2pdirectory.WifiDirectComingleDirectory import {
6      ownerLoc      :: int -> loc.
7      lookupIP      :: loc -> string.
8      retrieveDir    :: int -> {(loc,string,string)}.
9  }
10
11 predicate startOwner  :: string -> trigger.
12 predicate startMember :: string -> trigger.
13 predicate quit        :: trigger.
14 predicate connect     :: string -> trigger.
15
16 predicate joined      :: loc -> fact.
17 predicate member      :: string -> fact.
18 predicate owner       :: string -> fact.
19 predicate joinRequest :: (string,string,loc) -> fact.
20 predicate exit        :: loc -> fact.
21
22 predicate added       :: (loc,string,string) -> actuator.
23 predicate removed     :: loc -> actuator.
24 predicate connected   :: actuator.
25 predicate ownerQuit   :: actuator.
26 predicate deleteDir   :: actuator.
27
28 rule owner  :: [O]startOwner(C) --o [O]owner(C), [O]joined(O).
29 rule member :: [M]startMember(C) --o [M]member(C).
30
31 rule connect :: [M]member(C) \ [M]connect(N)
32     --o [O]joinRequest(C,N,M) where O = ownerLoc().
33
34 rule join :: [O]owner(C), {[O]joined(M')|M'->Ms},
35     \ [O]joinRequest(C,N,M) | notIn(M,Ms)
36     --o {[M']added(D)|M'<-Ms}, {[M]added(D')|D'<-Ds},
37     [M]added(D), [O]joined(M), [M]connected()
38     where IP = lookupIP(M), D = (M,IP,N), Ds = retrieveDir().
39
40 rule quitO :: [O]owner(C), [O]quit(), {[O]joined(M)|M->Ms}
41     --o {[M]ownerQuit()|M<-Ms}.
42
43 rule quitM :: {[O]joined(M')|M'->Ms.not(M' = M)}
44     \ [M]member(C), [M]quit(), [O]joined(M)
45     --o {[M']removed(M)|M'<-Ms}, [M]deleteDir().

```

Figure 15: WiFi-Direct Directory

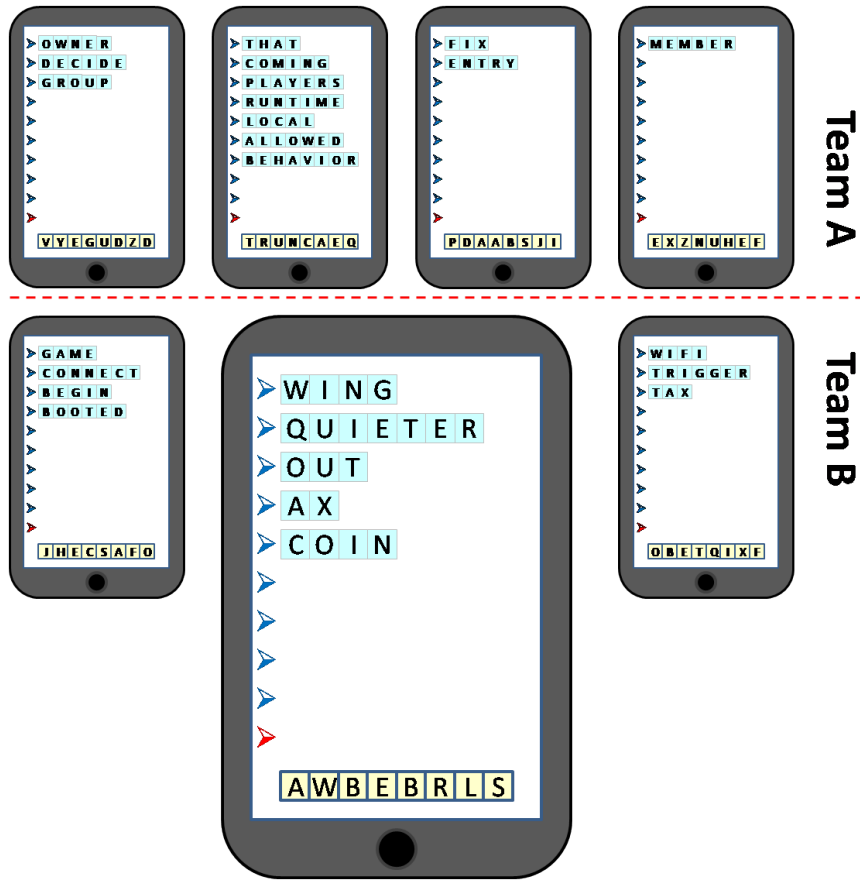


Figure 16: Swarbble, Mid-play Screen Shot

([M]deleteDir()).

In our current implementation, each Comingle runtime bootstraps an instance of this Comingle program, to maintain an active directory of the IP addresses of its peers. In all, this peer-to-peer directory Comingle application is written in 53 lines of Comingle codes (shown in Figure 15) and 154 lines of Java codes (in one Java class). This is a significant simplification from an earlier implementation written purely in Java (also available from our GitHub repository), which is had 694 lines of Java codes (in 10 Java classes).

## 5.4 Swarbble

Swarbble is a distributed word game loosely inspired to Scrabble. The game is played by two-teams of smartphone-toting players. A bird-eye view of a game of Swarbble is shown in Figure 16. Each player has a rack of randomly generated letters at the bottom of his or her screen, and a series of words in the top part. During the game, a player inserts letters from the rack at the beginning, end, or between any two letters of any of those words — he/she can also make an entirely new word on an empty row. If a word extended in this way is valid, the player can throw it at an opponent, thereby freeing a slot in his/her screen and filling a free slot on that opponent’s screen. The goal of the game is to flood the screen of each player in the other team with words: a player is booted out of the game when every slot in his/her screen has a word; a team wins when all the players on the other team are booted out. Players on the same team can rely on one another to prevent a member from being booted out: a player overwhelmed with words can seek help from his teammates, who can transfer one of his words, selected at random, to her own screen.

```

1  module comingle.lib.ExtLib import {
2    size :: {A} -> int.
3  }
4
5  predicate initGame    :: ({loc},{loc},int) -> trigger.
6  predicate throwNew   :: (loc,string) -> trigger.
7  predicate throwOld   :: (loc,string,string) -> trigger.
8  predicate relief     :: loc -> trigger.
9
10 predicate setup      :: ({loc},{loc},int) -> fact.
11 predicate checkWords :: fact.
12 predicate checkWin  :: fact.
13 predicate word      :: string -> fact.
14 predicate max       :: int -> fact.
15 predicate ally     :: loc -> fact.
16 predicate opp      :: loc -> fact.
17
18 predicate topUpRack  :: actuator.
19 predicate helpMe    :: loc -> actuator.
20 predicate refresh   :: actuator.
21 predicate win       :: actuator.
22 predicate booted    :: actuator.

```

Figure 17: Swarbble (Declarations)

Figure 18 shows the Comingle program that orchestrates the distributed behavior of Swarbble. A node  $I$  initializes the game by selecting the maximum number  $M$  of words that can appear on a player’s screen before he/she is booted (this act as the level of the game) and the two teams  $P_s$  and  $Q_s$  out ( $I$  may be in one of them, or may act as a disinterested third-party). This information is entered into  $I$ ’s rewrite state as the trigger  $[I] \text{initGame}(P_s, Q_s, M)$ , which enables rule `init`. This rule simply sends the facts  $[A] \text{setup}(A_s, B_s, M)$  to each player  $A$  in  $P_s$  and  $Q_s$ , where  $A_s$  are the other players on  $A$ ’s team and  $B_s$  are the players on the other team. Notice the syntactic shortcut  $P_s - \{P\}$  which computes the multiset obtained from  $P_s$  by removing element  $P$  (if present).

Each fact  $[X] \text{setup}(A_s, O_s, M)$  is used in rule `setup` to initialize the device of player  $X$ . It adds the facts  $[X] \text{ally}(A)$  and  $[X] \text{opp}(O)$  for each teammate  $A$  in  $A_s$  and for each opponent  $O$  in  $O_s$ , respectively, and records the word limit  $M$  in fact  $[X] \text{max}(M)$ . The actuator  $[X] \text{topoffRack}()$  randomly selects letters to complete the rack of player  $X$  (since this is the beginning of the game, this has the effect of generating  $X$ ’s entire rack). The actuator  $[X] \text{refresh}()$  invokes a screen refresh method that displays  $X$ ’s current list of words (which is initially empty), its teammates and its adversaries. When invoked, this method is able to notice changes in these lists, rendering them with appropriate animations.<sup>2</sup>

The next rule, `throwNew`, is executed once a player  $X$  has generated a new word from the letters in his or her rack and has performed the gesture to throw it at an opponent  $O$ . This has the effect of depositing the trigger  $[X] \text{throwNew}(O, \text{NewWd})$  in  $X$ ’s rewriting state. The body of the rule inserts the facts  $[O] \text{word}(\text{NewWd})$  and  $[O] \text{checkWords}()$  and the actuator  $[O] \text{refresh}()$  into  $O$ ’s rewriting state. The first fact makes the new word  $\text{NewWd}$  available to  $O$  while the second is used to determine if  $O$  is to be booted out or can call for help (see below). Rule `throwOld` is used similarly when a player  $X$  extends an existing word  $\text{Wd}$  and throws it to opponent  $O$ : the trigger is now  $[X] \text{throwOld}(O, \text{Wd}, \text{NewWd})$ , word  $\text{Wd}$  needs to be consumed from  $X$ ’s rewriting state, and the actuator  $[X] \text{refresh}()$  is added to assess  $X$ ’s state of play.

A fact  $[X] \text{checkWords}()$  is used by rules `allGood`, `help` and `boot` depending on the number  $w$  of words at

<sup>2</sup>The current implementation of Comingle allows the application runtime to access collections of facts maintained by the Comingle runtime. These collections can be attached to UI display interfaces of the Android SDK for example by using `ListFragment`.



```

23 rule init :: [I]initGame(Ps,Qs,M)
24   --o {[P]setup(Ps-{P},Qs,M) | P <- Ps},
25     {[Q]setup(Qs-{Q},Ps,M) | Q <- Qs}.
26
27 rule setup :: [X]setup(As,Os,M)
28   --o {[X]opp(O) | O <- Os}, {[X]ally(A) | A <- As}, [X]max(M),
29     [X]topoffRack(), [X]refresh().
30
31 rule throwNew :: [X]opp(O) \ [X]throwNew(O,NewWd)
32   --o [O]word(NewWd), [O]checkWords(),
33     [X]topoffRack(), [O]refresh().
34
35 rule throwOld :: [X]opp(O) \ [X]throwOld(O,Wd,NewWd), [X]word(Wd)
36   --o [O]word(NewWd), [O]checkWords(), [X]checkWords(),
37     [X]topoffRack(), [X]refresh(), [O]refresh().
38
39 rule allGood :: {[X]word(Wd) | Wd -> Wds}, [X]max(M)
40   \ [X]checkWords() | size(Wds) < M-3
41   --o 1.
42
43 rule help :: {[X]ally(A) | A -> As}, {[X]word(Wd) | Wd -> Wds},
44   [X]max(M)
45   \ [X]checkWords() | M-3 <= size(Wds), size(Wds) < M
46   --o {[A]helpMe(X) | A <- As}.
47
48 rule relief :: [A]ally(X) \ [A]relief(X), [X]word(Wd)
49   --o [A]word(Wd), [X]checkWords(), [A]checkWords(),
50     [X]refresh(), [A]refresh().
51
52 rule boot :: {[X]ally(A) | A -> As}, {[X]opp(O) | O -> Os},
53   {[X]word(Wd) | Wd -> Wds}, [X]max(M),
54   [X]checkWords() | size(Wds) >= M
55   --o [X]booted(), {[A]refresh() | A <- As},
56     {[O]refresh(), [O]checkWin() | O <- Os}.
57
58 rule win :: [X]checkWin(), {[X]opp(O) | O -> Os} | size(Os) = 0
59   --o [X]win().

```

Figure 18: Swarbble (Rules)

X in relation to the maximum M held in fact  $[X]_{\max}(M)$ . If  $w$  is less than  $M-3$ , rule `allGood` simply discards it. If  $w$  is between  $M-3$  and  $M$ , rule `help` to ask X's teammates for help by depositing the actuator `helpMe(X)` in each of their states. If a teammate A decides to help, the trigger  $[A]_{\text{relief}}(X)$  will be deposited in its state. Rule `relief` will pick a word  $w_d$  at random from X and transfer it to A's state. Both players will be issued new `checkWords()` facts and `refresh()` actuators.

If the number of words at X is greater or equal to the maximum allowed, rule `boot` arranges for X's demise by consuming all game facts at X, and issuing triggers  $[X]_{\text{booted}}()$  at X and `refresh()` at each teammate A and opponent O. The first actuator will display a notice telling X it has been booted from the game. This rule also deposits a fact  $[O]_{\text{checkWin}}()$  at each opponent.

The last rule, `win`, uses the fact  $[X]_{\text{checkWin}}()$  to declare X's team the winner whenever all the players in the opposing team have been booted. The trigger  $[X]_{\text{win}}()$  displays an appropriate animation.

## 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 [3] extended Meld with comprehension patterns and linearity, but refocused it on distributed programming of multicore architectures.

Sifteo [14] 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 [6], 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 four distributed games orchestrated by Comingle and running on Android mobile devices. 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, and possibly target platforms other than Android. In particular, we want to explicitly support application rendering subroutines by maintaining dynamic "views" that can be directly attached to user interface fragments (e.g., `ListFragment` of the Android SDK).

## References

- [1] M.P. Ashley-Rollman, P. Lee, S.C. Goldstein, P. Pillai, and J. D. Campbell. A Language for Large Ensembles of Independently Executing Nodes. In *ICLP '09*, July 2009.

- [2] P. Chaganti. *Google Web Toolkit GWT Java AJAX Programming: A Step-by-step to Google Web Toolkit for Creating Ajax Applications Fast*. Packt Publishing, 2007.
- [3] F. Cruz, R. Rocha, S.C. Goldstein, and F. Pfenning. A linear logic programming language for concurrent programming over graph structures. In *ICLP'14*, Vienna, Austria, 2014.
- [4] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS'08/ETAPS'08*, pages 337–340, Berlin, Heidelberg, 2008. Springer.
- [5] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI04*. USENIX Association, 2004.
- [6] T. Frühwirth. Constraint Handling Rules. In *Constraint Prog.*, pages 90–107, 1994.
- [7] New York Google. Chrome Racer, A Chrome Experiment. <http://www.chrome.com/racer>, 2013.
- [8] B.T. Kirby, M. Ashley-Rollman, and S.C. Goldstein. Blinky blocks: A physical ensemble programming platform. In *CHI'11*, pages 1111–1116, New York, NY, USA, 2011. ACM.
- [9] E.S.L. Lam and I. Cervesato. Decentralized Execution of Constraint Handling Rules for Ensembles. In *PPDP'13*, pages 205–216, Madrid, Spain, 2013.
- [10] E.S.L. Lam and I. Cervesato. Constraint Handling Rules with Multiset Comprehension Patterns. In *CHR'14*, 2014.
- [11] E.S.L. Lam and I. Cervesato. Optimized Compilation of Multiset Rewriting with Comprehensions. In *APLAS'14*, pages 19–38. Springer LNCS 8858, 2014.
- [12] E.S.L. Lam and I. Cervesato. Optimized Compilation of Multiset Rewriting with Comprehensions (Full-Version). Technical Report CMU-CS-14-119, Carnegie Mellon, June 2014.
- [13] E.S.L. Lam and I. Cervesato. Reasoning about Set Comprehension. In *SMT'14*, 2014.
- [14] D. Merrill and J. Kalanithi. Sifteo, Interactive Game Cubes. <https://www.sifteo.com/cubes>, 2009.
- [15] P.J. Stuckey, M. Sulzmann, and J. Wazny. Interactive Type Debugging in Haskell. In *Haskell'03*, pages 72–83, New York, NY, USA, 2003. ACM.

## A Compiled Drag Racing Example

In this appendix, we show some key fragments of the Java class generated by the compilation of the Drag Racing example. This Java class, `Dragracing` extends the `RewriteMachine` class that implements the Comingle runtime. The `DragRacing` class specializes this runtime to execute decentralized multiset rewriting as specified by the Drag Racing Comingle program. The complete code generated by the compiler is available at [https://github.com/sllam/comingle/tree/master/android\\_apps/CoMingleDragRacing](https://github.com/sllam/comingle/tree/master/android_apps/CoMingleDragRacing) and consists of 1808 lines.

Figure 19 shows a fragment of the `DragRacing` Java class related to the representation of facts and the triggering and actuation interfaces. Each predicate (standard, trigger or actuator) of the Drag racing Comingle program is represented as an inner Java class that extends the `DragracingFact` class. Figure 19 shows the inner classes `InitRace` and `StartRace` that represents the predicates of the same name in the Comingle program. The application runtime interacts with the Drag Racing rewriting machine by adding triggers by interface methods such as `addInitRace` also shown in Figure 19. These methods add a new fact of the respective kind and its input parameters to an internal queue. When such a fact is processed, the compiled rules are used to determine the incremental changes to the rewriting state it determines.

Figure 20 shows a fragment of such a compiled rule. The method `execute_starttrace_join_ordering_1` implements a matching routine that is invoked by the addition of the trigger fact `StartRace`. This matching routine

```

1 public class Dragracing extends RewriteMachine {
2
3     abstract class DragracingFact extends Fact { ... }
4     public class InitRace extends DragracingFact { ... }
5     public class StartRace extends DragracingFact { ... }
6
7     ...
8
9     public void addInitRace(LinkedList<Integer> arg1) {
10        intro( new InitRace(myLocation, arg1) );
11    }
12    public void addStartRace() {
13        intro( new Go(myLocation) );
14    }
15    public void addSendTap() {
16        intro( new SendTap(myLocation) );
17    }
18    public void addExiting(int arg1) {
19        intro( new Exiting(myLocation, arg1) );
20    }
21
22    ...
23
24    public void setRenderTrackActuator
25        (ActuatorAction<LinkedList<Integer>> action) {
26        setActuator(Actuations.rendertrack, action);
27    }
28    public void setReleaseActuator(ActuatorAction<Unit> action) {
29        setActuator(Actuations.release, action);
30    }
31    public void setRecvTapActuator(ActuatorAction<Integer> action) {
32        setActuator(Actuations.recvtap, action);
33    }
34    public void setHasActuator(ActuatorAction<Integer> action) {
35        setActuator(Actuations.has, action);
36    }
37    public void setDecWinnerActuator(ActuatorAction<Integer> action) {
38        setActuator(Actuations.decwinner, action);
39    }
40
41    ...
42 }

```

Figure 19: Drag Racing Compilation: Facts, Triggers and Actuators

```

1 public class Drag Racing extends RewriteMachine {
2     ...
3     // rule start :: [X]all(Ps) [X]startRace() --o {[P]release()|P<-Ps}.
4     ...
5     protected boolean execute_startrace_join_ordering_1(StartRace act) {
6         int x;
7         SimpMultiset<Integer> ls;
8         int l;
9         // Join Task: Active #H0 [X]startRace()
10        x = act.loc;
11        // Join Task: LookupAtom #H1 8:0:hash<[+]all(-)|.> X [X]all(Ls)
12        StoreIter<All> candidates_1 =
13            all_store_0.lookup_candidates(index0All(x));
14        All cand_1 = candidates_1.get_next_alive();
15        while(cand_1 != null) {
16            int x1;
17            x1 = cand_1.loc;
18            ls = cand_1.arg1;
19            if (true) {
20                // Join Task: DeleteHead #H0
21                // H0 is active and monotone, no delete required
22                // Join Task: IntroCompre Remote NoPrior Mono L Ls [L]release()
23                SimpMultiset<Integer> comp_0 = ls;
24                for(int idx=0; idx<comp_0.size(); idx++) {
25                    l = comp_0.get(idx);
26                    send( new Release(l) );
27                }
28                startrace_rule_count++;
29                return false;
30            }
31            cand_1 = candidates_1.get_next_alive();
32        }
33        return true;
34    }
35    ...
36 }

```

Figure 20: Drag Racing Compilation: Rule Compilation Example

attempts to apply the `start` rule of the Drag Racing Comingle program by matching the `StartRace` fact to a compatible `all` fact. A Comingle rule is compiled into a set of methods like this, each corresponding to one of the rule's head. Details of this compilation scheme is out of the scope of this technical report, but are available in [11, 12].