Programmable Orchestration of Time-Synchronized Events Across Decentralized Android Ensembles

Edmund S. L. Lam Iliano Cervesato Ali Elgazar
Carnegie Mellon University Qatar
Email: {sllam, iliano, aee}@cmu.edu

***Abstract*—Orchestrating a time sensitive computation across an ad hoc ensemble of Android devices is surprisingly challenging in spite of the OS's support for automated network time synchronization. In fact, the lack of access to programmatic nor end-user control pushes the responsibility of implementing fine-grained time synchronization to the application development level. In this paper, we extend CoMingle, a distributed logic programming framework designed for orchestrating Android ensembles, with explicit time annotations and built-in network time synchronization support. The result is a powerful programmable orchestration framework for developing apps that exhibits time-synchronized events across an ensemble of Android devices.**

I. Introduction

In spite of standard network time services such as NTP, NITZ and GPS installed by default to automatically synchronize local clocks, an ad hoc collection of Android mobile devices often has members that are mutually out-of-sync by seconds, sometimes minutes — this is due to time drift, faulty software/hardware or infrequent syncing. This makes it extremely challenging to implement mobile applications that rely on tightly time-synchronized events across multiple Android devices. And this is not likely to change any time soon: Google has announced that programmatic or end-user control over network time synchronization parameters (e.g., manual calibration, frequency) will not be provided in standard (non rooted) Android distributions [6]. This forces the Android application developer to take on the responsibility of implementing any needed finer-grained time synchronization. Time synchronization across distributed systems is a well-studied problem [12], [9], [14] and, in the context of sensor networks, numerous practical synchronization algorithm are available (e.g., [3], [5], [13]). However, deploying such specialized solutions in each application is tedious and time consuming.

In this paper, we develop a high-level programmable orchestration mechanism that allows application developers to express time-synchronized events and execute them across an ensemble of Android devices. This approach extends CoMingle [11] with *time annotations* that express distributed timing requirements over multiple devices. CoMingle is a programming framework aimed at simplifying the development of distributed applications across an ensemble of Android devices. To support this new language construct, the CoMingle runtime is augmented to seamlessly synchronize devices in the ensemble. This provides developers with the means of rapidly programming complex communication patterns that orchestrate time-synchronized events across devices. Doing so lowers the technical bar to developing Android applications that feature time-synchronized events, especially UI events that users of devices in close proximity to each other can observe. Altogether, this papers makes the following contributions:

- We define an extended version of CoMingle that introduces time annotations to the language and expresses timing obligations in its semantics.
- We show how timed events specified in CoMingle can be synchronized using traditional time-synchronization protocols and integrate a simplified instance of TPSN [5] into the CoMingle runtime system. This approach works for *any* collection of Android devices.
- We show two proof-of-concept distributed Android applications that utilize time-synchronized events orchestrated by CoMingle. The source code and pre-compiled Android package are available at *https://github.com/sllam/comingle*.

The rest of the paper is organized as follows: we introduce CoMingle through an example in Section II and then formally in Section III. Section IV describes our time synchronization scheme, while in Section V we present two case studies that use synchronized events. We discuss related work in Section VI and conclude in Section VII.

II. An Example

Drag Racing is a simple multi-player game inspired by Chrome Racer [7]. A number of players compete to reach the finish line of a linear car 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 1. Figure 1 also shows the CoMingle program that orchestrates the Drag Racing game. 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 a predicate name and $\vec{t}$ are terms. A CoMingle program consists of a set of rules, each of the form $r :: \overline{H}_p \backslash \overline{H}_s \multimap \overline{B}$. Such a rule describes a possible transformation of the state of the ensemble: if the ensemble contains located facts matching $\overline{H}_p$ and $\overline{H}_s$, the application of the rule replaces the facts identified by $\overline{H}_s$ with the facts specified by $\overline{B}$. The expressions $\overline{H}_p$ and $\overline{H}_s$ are called heads of the rule and $\overline{B}$ is its body. Auxiliary computation is carried out in an optional **where** clause. See Section III for a detailed definition.

An initial configuration such as the one in Figure 1 is generated when rule `init` is executed. Its head is the fact `[I]initRace(Ls)`, where node `I` holds the initial segment of the track and `Ls` lists all locations participating in the game (including `I`). Several actions need to take place at

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

Fig. 1. Drag Racing, a racing game inspired by Chrome Racer

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 line `(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 line `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 expressions `{[A]next(B)|(A,B)<-Cs}` and `[E]last()`, respectively. The expression `{[A]next(B)|(A,B)<-Cs}` is a *comprehension pattern* and stands for a multiset of facts `[A]next(B)` for each pair `(A,B)` in the multiset `Cs`. 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)`).

Underlined predicates (e.g., `initRace`) identify *trigger facts* and act as interfaces from a device's local computation. Specifically, a fact `[ℓ]initRace(Ls)` is entered into the rewriting state by a local program at ℓ and used to trigger the initialization of the game. Trigger facts are only allowed to appear in the heads of a rule. Dually, facts like `[ℓ]renderTrack(Ls)` and `[ℓ]has(P)` are *actuator facts*, generated by the rewriting process for the purpose of starting local computations at ℓ. We underline actuator facts with dashed lines. 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 instance, the actuator `[ℓ]has(P)` is concretely implemented as a Java callback function that calls ℓ's local UI thread to render player `P`'s sprite on ℓ's screen.

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]stRace()` 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]exit(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)`). Fact `[Y]at(X)` gets consumed. If instead `X` holds the final segment of the track (`[X]last()`) when the trigger `[X]exit(Y)` materializes, `Y`'s victory is broadcast to all participating locations (`{[P]decWinner(Y)|P<-Ps}`).

Time-synchronized Start: While the implementation of Drag Racing in Figure 1 runs reasonably well, it does not guarantee that the actuator `release()` will be processed at the same time on each device,[1] which may give some players an advantage. We address this concern by introducing time annotations in body facts to constrain when they should be acted upon. Rule `start` could then be updated as follows (in yellow):

```
1 rule start :: [X]all(Ps) \ [X]stRace()
2               --o {[P]release()@T|P<-Ps}
3                   where T=now()+500.
```

Each participating device `P` is sent an *event* `release()@T`. The argument `T` is a precise time in the future where the actuator `release()` shall be processed. Time `T` is set as the current time `now()` at location `X` plus a small delay (500ms), giving enough time for all messages to be delivered to each device `P` in `Ps`. Now, the local clock at `P` may be out of sync relative to `X` (by seconds in our experiments). Therefore, having each `P` process `release()` at time `T` (on `X`) requires them to compute a local time offset relative to `X` (or any common time reference). Section IV discusses in details a synchronization mechanism to achieve this effect.

While precise time synchronization is not critical for Drag Racing, Section V examines two applications whose functionality relies on events being tightly synchronized across devices.

III. CoMingle with Explicit Time

We now describe the abstract semantics of CoMingle. 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. In the following, updates to the syntax and semantics of CoMingle [11] are highlighted in yellow.

[1] With the implementation in Figure 1, time difference between invocations of the actuators, due to the small time delays in LAN or WiFi-direct communications, are indeed occasionally observable.

Locations: ℓ Terms: t Guards: g Time expressions: c Standard / trigger / actuator predicates: p_s, p_t, p_a

$$\begin{array}{llll}\text{Standard facts} & F_s ::= [\ell]\,p_s(\vec{t}) & \text{Triggers} \quad F_t ::= [\ell]\,p_t(\vec{t}) & \text{Actuators} \quad F_a ::= [\ell]\,p_a(\vec{t})\end{array}$$

$$\text{Facts} \quad f, F ::= F_s \mid F_t \mid F_a \qquad \text{Events} \quad E ::= F_a@c \mid F_s@c$$

$$\begin{array}{llll}\text{Head atoms} & h ::= F_s \mid F_t & \text{Body atoms} & b ::= F_s \mid F_a \mid E \\ \text{Head expressions} & H ::= h \mid \wr h \mid g \int_{\vec{x}\in t} & \text{Body expressions} & B ::= b \mid \wr b \mid g \int_{\vec{x}\in t} \\ \text{CoMingle rule} & R ::= \overline{H} \setminus \overline{H} \mid g \multimap \overline{B} & \text{CoMingle program} & \mathcal{P} ::= \overline{R}\end{array}$$

$$\text{Rewriting state} \quad St ::= \overline{F} \qquad \text{Local state: } [\ell]\psi \qquad \text{Application state} \quad \Psi ::= \overline{[\ell]\psi} \qquad \text{CoMingle state} \quad \Theta ::= \langle St; \Psi; \overline{E}\rangle$$

Fig. 2. Abstract Syntax and Runtime Artifacts of CoMingle with Time Annotations

A. Abstract Syntax

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. Facts also carry a tuple $\vec{t}$ of *terms*. Ground facts, which do not contain free variables, are denoted F_s, F_a and F_t. The abstract semantics of CoMingle is largely agnostic to the specific language of terms. An *event* E is an actuator $f_a@c$ or standard fact $f_s@c$ annotated with a time expression c.

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 or events (f_s or f_a or E). 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. 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 | x -> t. g}` in rule heads and `{f | x <- t. g}` in a rule body, where the direction of the arrow acts as a reminder of the flow of information. Comprehensions with the same range can be combined. 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 user-defined Boolean functions. Some guards are written as **where** clauses.

B. Overview of the Abstract Semantics

We describe the computation of a CoMingle system by means of a small-step transition semantics. Its basic judgment has 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 $\langle St; \Psi; \overline{E}\rangle$. 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). As we will see, a local computation step transforms the application state Ψ but can also insert triggers into the rewriting state and consume actuators from it. These run-time artifacts are formally defined at the bottom of Figure 2. The third component, the *scheduled events* $\overline{E}$, is a collection of events that await execution. In this semantics, the local time at ℓ is retrieved by the meta-level operation $\texttt{clock}(\ell)$. In this paper, we focus on modeling the fulfillment of time obligations, ensuring the timely execution of the scheduled events.

In Section III-C, we introduce auxiliary judgments and in Section III-D we combine them into the overall abstract semantics of CoMingle, defined by the state transition $\mathcal{P} \rhd \Theta \mapsto \Theta'$ (see Figure 6 for a preview). The CoMingle prototype is based on a *concrete semantics* [10] that efficiently implements the abstract semantics in this section.

C. Matching, Processing Rule Body and Time Obligations

The application of a CoMingle rule $\overline{H}_p \setminus \overline{H}_s \mid g \multimap \overline{B}$ to a state $\langle St; \Psi; \overline{E}\rangle$ involves two main operations: identifying fragments of the rewriting state St that match the rule heads $\overline{H}_p$ and $\overline{H}_s$, and replacing $\overline{H}_s$ in St with the corresponding instance of the body $\overline{B}$. We now extend the formalization of these operations in [10] to account for time obligations.

Matching Rule Heads: Let $\overline{H}$ be a (preserved or consumed) rule head without free variables. 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 $\overline{H} \triangleq St^+$ defined in the top part of Figure 3. Rules l_{mset-*} partition St^+ into fragments to be matched by each atom in $\overline{H}$: plain facts F must occur identically (rule l_{fact}) while for comprehension atoms $\wr f \mid g \int_{\vec{x}\in \overline{ts}}$ 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 l_{cp*}).

$$\frac{\overline{H} \triangleq St \quad H \triangleq St'}{\overline{H}, H \triangleq St, St'} (\mathsf{l}_{mset1}) \qquad \frac{}{\varnothing \triangleq \varnothing} (\mathsf{l}_{mset2})$$

$$\frac{[\vec{t}/\vec{x}]F' \triangleq F \quad \models [\vec{t}/\vec{x}]g \quad \wr F' \mid g \int_{\vec{x} \in \overline{ts}} \triangleq St}{\wr F' \mid g \int_{\vec{x} \in \vec{t}, \overline{ts}} \triangleq St, F} (\mathsf{l}_{cp1})$$

$$\frac{}{\wr F \mid g \int_{\vec{x} \in \varnothing} \triangleq \varnothing} (\mathsf{l}_{cp2}) \qquad \frac{}{F \triangleq F} (\mathsf{l}_{fact})$$

$$\frac{\overline{H} \triangleq^{\neg} St \quad H \triangleq^{\neg} St}{\overline{H}, H \triangleq^{\neg} St} (\mathsf{l}^{\neg}_{mset1}) \qquad \frac{}{\varnothing \triangleq^{\neg} St} (\mathsf{l}^{\neg}_{mset2})$$

$$\frac{F \not\sqsubseteq \wr F' \mid g \int_{\vec{x} \in ts} \quad \wr F' \mid g \int_{\vec{x} \in ts} \triangleq^{\neg} St}{\wr F' \mid g \int_{\vec{x} \in ts} \triangleq^{\neg} St, F} (\mathsf{l}^{\neg}_{cp1})$$

$$\frac{}{\wr F \mid g \int_{\vec{x} \in ts} \triangleq^{\neg} \varnothing} (\mathsf{l}^{\neg}_{cp2}) \qquad \frac{}{F \triangleq^{\neg} St} (\mathsf{l}^{\neg}_{fact})$$

Subsumption:
$F \sqsubseteq \wr F' \mid g \int_{\vec{x} \in ts}$ iff $F = \theta F'$ and $\models \theta g$ for some $\theta = [\vec{t}/\vec{x}]$

Fig. 3. Matching a Rule Head: $\overline{H} \triangleq St \quad \overline{H} \triangleq^{\neg} St$

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 $\overline{H} \triangleq^{\neg} St^{-}$ in the bottom part of Figure 3. Rules $\mathsf{l}^{\neg}_{mset*}$ test each individual atom and rule $\mathsf{l}^{\neg}_{fact}$ ignores facts. Rules $\mathsf{l}^{\neg}_{cp*}$ deal with comprehensions $\wr f \mid g \int_{\vec{x} \in \overline{ts}}$: they check that no fact in St^{-} matches any instance of f while satisfying g.

Processing Rule Bodies: Applying a CoMingle rule involves extending the rewriting state with the facts and events corresponding to its body. This operation, captured by the judgment $\overline{B} \gg \langle St; \overline{E} \rangle$, is specified in Figure 4 for a closed body $\overline{B}$. Rules r_{mset*} go through $\overline{B}$. Atomic facts F are added immediately to St (rule r_{fact}) while events E are added to $\overline{E}$ (rule r_{event}). Comprehension atoms $\wr f \mid g \int_{\vec{x} \in \overline{ts}}$ need to be *unfolded* (rules r_{c*}): for every item $\vec{t}$ in $\overline{ts}$ that satisfies the guard g, the corresponding instance $[\vec{t}/\vec{x}]f$ is added to either St or $\overline{E}$; instances that do not satisfy g are discarded.

Fulfilling Time Obligations: Our timed extension of CoMingle needs to make scheduled events available for rewriting and actuation when their time has come. The judgment $\Psi \vdash_{\delta} \overline{E}$, defined in Figure 5, supports this behavior by checking that each event $[\ell]f@c$ in $\overline{E}$ is scheduled for execution at a future time, according to location ℓ's local clock ($\mathtt{clock}(\ell) < c$).

$$\frac{\mathtt{clock}(\ell) < c \quad \Psi, [\ell]\psi \vdash_{\delta} \overline{E}}{\Psi, [\ell]\psi \vdash_{\delta} \overline{E}, [\ell]f@c} (\delta_{\mathsf{mset1}}) \qquad \frac{}{\Psi \vdash_{\delta} \varnothing} (\delta_{\mathsf{mset2}})$$

Fig. 5. Time Obligation Fulfillment: $\Psi \vdash_{\delta} \overline{E}$

D. Abstract Semantics

Figure 6 defines the state transition $\mathcal{P} \triangleright \Theta \mapsto \Theta'$. There are three forms of state transitions. Rules rw_ev_act and rw_ev_std handle scheduled events $[\ell]F_a@c$ and $[\ell]F_s@c$, respectively. In rw_ev_act, an actuator event $[\ell]F_a@c$ is

$$\frac{\overline{B} \gg \langle St; \overline{E} \rangle \quad B \gg \langle St'; \overline{E'} \rangle}{\overline{B}, B \gg \langle St, St'; \overline{E}, \overline{E'} \rangle} (\mathsf{r}_{mset\text{-}1}) \qquad \frac{}{\varnothing \gg \langle \varnothing; \varnothing \rangle} (\mathsf{r}_{mset2})$$

$$\frac{}{F \gg \langle F; \varnothing \rangle} (\mathsf{r}_{fact}) \qquad \frac{}{E \gg \langle \varnothing; E \rangle} (\mathsf{r}_{event})$$

$$\frac{\models [\vec{t}/\vec{x}]g \quad [t/\vec{x}]b \gg \langle \overline{F}; \overline{E} \rangle \quad \wr b \mid g \int_{\vec{x} \in ts} \gg \langle St; \overline{E'} \rangle}{\wr b \mid g \int_{\vec{x} \in \vec{t}, ts} \gg \langle \overline{F}, St; \overline{E}, \overline{E'} \rangle} (\mathsf{r}_{c1})$$

$$\frac{\not\models [\vec{t}/\vec{x}]g \; \wr b \mid g \int_{\vec{x} \in ts} \gg \langle St; \overline{E} \rangle}{\wr b \mid g \int_{\vec{x} \in \vec{t}, ts} \gg \langle St; \overline{E} \rangle} (\mathsf{r}_{c2}) \qquad \frac{}{\wr b \mid g \int_{\vec{x} \in \varnothing} \gg \langle \varnothing; \varnothing \rangle} (\mathsf{r}_{c3})$$

Fig. 4. Processing a Rule Body: $\overline{B} \gg \langle St; \overline{E} \rangle$

moved to the rewrite state St only if its time c has arrived ($c \leq \mathtt{clock}(\ell)$), and similarly for standard events in rule rw_ev_std. This allows the remaining rules to expect that all events be scheduled in the future ($\Psi \vdash_{\delta} \overline{E}$). Because c often refers to the time at a different location, how close that intended time is to $\mathtt{clock}(\ell)$ depends on how tightly synchronized the ensemble is, which is examined in Section IV.

Rule rw_ens 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 ($\theta\overline{H}_p \triangleq St_p$ and $\theta\overline{H}_s \triangleq St_s$), while the remaining fragment St must be free of residual matches ($\theta(\overline{H}_p, \overline{H}_s) \triangleq^{\neg} St$). The rule body instance $\theta\overline{B}$ is then unfolded ($\theta\overline{B} \gg \langle St_b; \overline{E}_b \rangle$) into St_b which replaces St_s in the rewriting state and $\overline{E}_b$ that is added to $\overline{E}$. An important side condition is $\Psi \vdash_{\delta} \overline{E}$, which dictates that this transition is only possible if all events $\overline{E}$ are in the future, thereby prioritizing rules rw_ev_*.

Rewriting steps defined by rule rw_ens 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 $\langle \mathcal{A}; \psi \rangle \mapsto_{\ell} \langle \psi'; \mathcal{T} \rangle$ 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 6 incorporates 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 triggers back 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. Like rule rw_ens, rule rw_loc applies only if the events $\overline{E}$ are in the future ($\Psi \vdash_{\delta} \overline{E}$). Note that $\mathcal{A}$ may be empty: a transition step taken with an empty $\mathcal{A}$ corresponds to an internal computation at location ℓ not initiated within the rewriting state. This includes the ticking of the internal clock ($\mathtt{clock}(\ell)$).

$$\text{Local transitions:} \quad \langle \mathcal{A}; \psi \rangle \mapsto_\ell \langle \mathcal{T}; \psi' \rangle \qquad \text{Local Clock:} \quad \texttt{clock}(\ell)$$

$$\frac{\Psi \vdash_\delta \overline{E} \quad (\overline{H}_p \setminus \overline{H}_s \mid g \multimap \overline{B}) \in \mathcal{P} \quad \models \theta g \quad \theta\overline{H}_p \triangleq St_p \quad \theta\overline{H}_s \triangleq St_s \quad \theta(\overline{H}_p, \overline{H}_s) \triangleq^\neg St \quad \theta\overline{B} \gg \langle St_b; \overline{E_b} \rangle}{\mathcal{P} \rhd \langle St_p, St_s, St; \Psi; \overline{E} \rangle \mapsto \langle St_p, St_b, St; \Psi; \overline{E}, \overline{E_b} \rangle} \text{(rw_ens)}$$

$$\frac{c \leq \texttt{clock}(\ell)}{\mathcal{P} \rhd \langle St; \Psi; \overline{E}, [\ell]F_s@c \rangle \mapsto \langle St, [\ell]F_s; \Psi; \overline{E} \rangle} \text{(rw_ev_std)}$$

$$\frac{\Psi \vdash_\delta \overline{E} \quad \langle \mathcal{A}; \psi \rangle \mapsto_\ell \langle \mathcal{T}; \psi' \rangle}{\mathcal{P} \rhd \langle St, [\ell]\mathcal{A}; \Psi, [\ell]\psi; \overline{E} \rangle \mapsto \langle St, [\ell]\mathcal{T}; \Psi, [\ell]\psi'; \overline{E} \rangle} \text{(rw_loc)}$$

$$\frac{\langle F_a; \psi \rangle \mapsto_\ell \langle \mathcal{T}; \psi' \rangle \quad c \leq \texttt{clock}(\ell)}{\mathcal{P} \rhd \langle St; \Psi, [l]\psi; \overline{E}, [\ell]F_a@c \rangle \mapsto \langle St, [\ell]\mathcal{T}; \Psi, [l]\psi'; \overline{E} \rangle} \text{(rw_ev_act)}$$

Fig. 6. Abstract Semantics of CoMingle: $\mathcal{P} \rhd \langle St; \Psi; \overline{E} \rangle \mapsto \langle St'; \Psi'; \overline{E'} \rangle$

IV. Implementing Event Synchronization

Given a set of events to be executed at some common time c, for example $\{[\ell]F_a@c | \ell \in L\}$, the task of guaranteeing that the mobile devices at locations $\ell \in L$ invoke their respective instance of actuator F_a simultaneously is far from straightforward. In fact, each device will schedule F_a at time c *based on its local clock* which may be off (sometimes by seconds) rather than a universal time. Android provides services such as NITZ or NTP to synchronize time and yet skews of one to two seconds are not uncommon. Google prevents third party apps from resetting system clocks, citing security concerns [6]. Therefore, in order to synchronize events, each device in an ensemble needs to determine the offset of its internal clock relative to some common *referential time*. We review two solutions we have implemented to compute this offset on each device.

Network Time Protocol (NTP) Synchronization: A solution to this problem is to have each device seek out a common referential time from some time synchronization service and compute the *time offset* of its own internal clock from this referential time. For instance, an initial prototype of our CoMingle runtime utilizes an open-source NTP library (specifically, Apache Commons Net library) for this purpose. Each device retrieves a referential time stamp via an NTP exchange and compares it with its own internal clock to compute its local time offset. Using NTP, however, imposes the requirement that each device of the ensemble have access to the Internet.

Local Referential Time Synchronization: To orchestrate event synchronization on decentralized mobile networks (e.g., WiFi-direct groups), we have implemented an alternative: rather than seeking an NTP referential time, the devices of the ensemble offset their internal clocks based on the local time of one of them. In order for each node to obtain a referential time from this node, we provide a synchronization protocol that accounts for propagation delays in the network. Synchronization protocols for sensor networks (e.g., [3], [5]) are directly applicable. We have implemented a simplified variant of the Timing-sync Protocol for Sensor Networks (TPSN) [5] for single-hop networks. This is sufficient for many applications as the current CoMingle supports only WiFi-direct or LAN connections, which correspond to single-hop network architectures.

The diagram in Figure 7 illustrates the simplified TPSN protocol we implemented to compute time offset. A device within the ensemble is chosen to act as the *referential time server*. All other devices are *referential time clients*. In this scheme, the clients compute their local time offset (δ_{offset}) through the following steps: (1) A client sends a message, identified by a randomly generated integer n, to the server indicating its intent to synchronize, and records the time T_1 when this request is sent. (2) Upon receiving this request, the server makes a note of the time T'_2 it receives the request and the time T'_3 it sends its response back to the client. The response contains these two time stamps, along with n. (3) Upon receiving the server's response, the client notes the time T_4. The offset time of the client (denoted δ_{offset}) is ideally computed by comparing T_1 with the hypothetical T'_1, the time at the server when the client read T_1. We approximate T'_1 by offsetting T'_2 with the propagation delay (δ_{Delay}), which can be estimated from the round trip time of the request and response. To make these estimates more reliable, our current implementation has each member compute the median of the time offsets over a number of synchronization polls. This simple scheme provides a synchronization protocol that performs well for the applications we have considered so far (most events were synchronized well within an imperceptible 10ms). In the future, more sophisticated protocols (e.g., [3], [13]) can be implemented to provide more precise time synchronization to the CoMingle runtime.

We have augmented the CoMingle runtime with routines that provide this fine-grained time synchronization. These time synchronization routines are activated when the CoMingle compiler determines the presence of time annotations in the source CoMingle program. Figure 8 illustrates our CoMingle framework (see also [11]). In our current implementation, if the devices are connected via WiFi-direct, the referential time server is the WiFi-direct group owner. If they are connected via a local area network, this role is assumed by the device which initiated the application. The default setting is for each

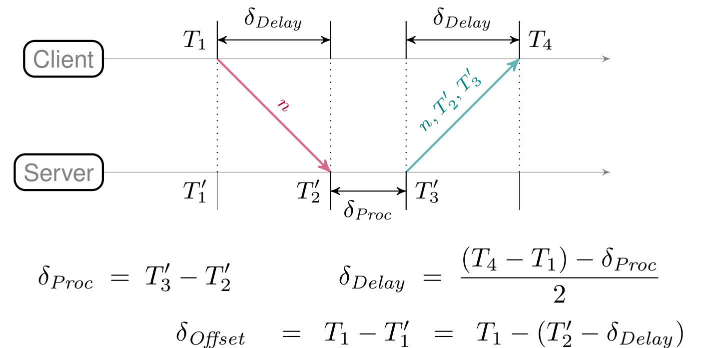


Fig. 7. Simplified TPSN

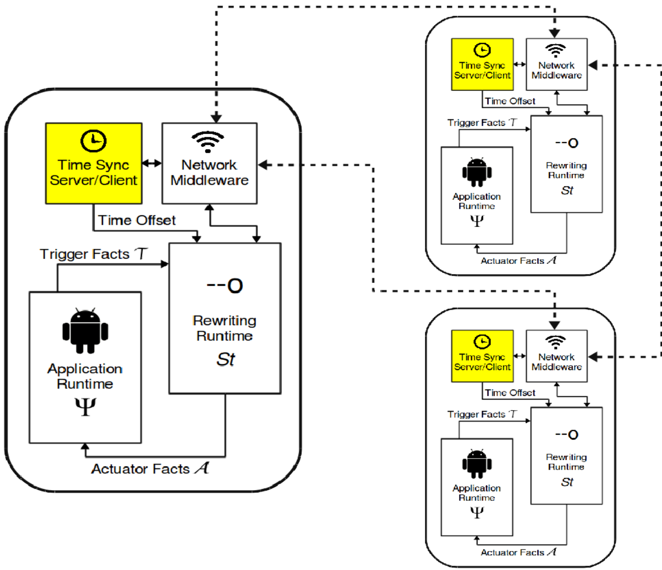


Fig. 8. CoMingle with Time Synchronization

```
1  rule dist :: [I]dist(Ns,Vs,Es)
2          --o [I]trans(Ns,0), [I]all(Vs),
3              {[V]source(I)|V<-Vs},
4              {[F]edge(T)|(F,T)<-Es}.
5
6  rule fwd :: {[X]edge(Y)|Y->Ys}\[X]trans(N:Ns,P)
7            | size(Ys)>0
8          --o [X]note(N,P),
9              {[Y]trans(Ns,P+1)|Y<-Ys}.
10 rule ret :: {[X]edge(Y)|Y->Ys}, [X]source(I)
11           \ [X]trans(N:Ns,P) | size(Ys)=0
12         --o [X]note(N,P), [I]trans(Ns,P+1).
13
14 rule end :: [I]source(I) \ {[_]trans([],_)},
15             {[X]trans(_:_,_)|X->Xs}
16           | size(Xs)=0 --o [I]completed().
17 rule play :: [I]all(Xs) \ [I]start(S),
18             {[X]note(N,P)|(X,N,P)->Ms,in(X,Xs)}
19       --o {[X]play(N)@(T+P*1000)|(X,N,P)<-Ms}
20       where T = now()+S.
```

Fig. 9. Musical Shares, Cooperative Acoustics across an Android Ensemble

client device to poll the server for referential time at the point of its entry into the CoMingle ensemble. Our CoMingle library includes a set of APIs to allow developers to control the period and frequency of such polling during the life-cycle of the CoMingle runtime.

V. Case Studies

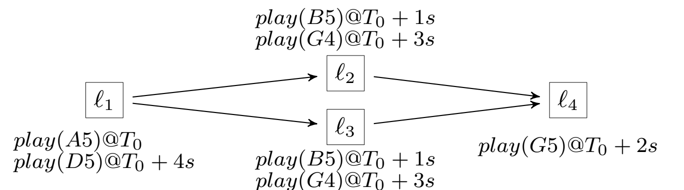
In this section, we show two examples of applications that rely on time annotations to synchronize distributed events.

A. Musical Shares is a CoMingle program that plays a musical score across an ensemble of participating Android devices. The node I that initiates the performance is given a sequence Ns of musical notes to play and an undirected graph (Vs, Es) which describes how the notes are to be distributed across the ensemble. In particular, starting from a source node $I \in Vs$, the musical score Ns is propagated across the edges of the graph. Each node (device) receives a subsequence Ns' of Ns, keeps the head N of Ns', records the position of N in the original score, and sends the tail of Ns' to all its outgoing edges. If a node has no outgoing edge, the tail of the sequence is returned to I. For simplicity, we assume that each note is to be played for one second at a time equal to its position in the score (if variable tempo is desired, each note can be distributed with information on how long it should be played).

Figure 9 displays the CoMingle program that orchestrates Musical Shares. Rule `dist` on lines 1–4 initiates the distribution of a musical score Ns from source location `I`, based on a graph `(Vs,Es)` over the ensemble. The trigger fact `[I]dist(Ns,Vs,Es)` initiates this process, in which the following happens: (1) source location `I` starts off the distribution with `trans(Ns,0)` and records all the locations involved (`all(Vs)`), (2) all nodes in `Vs` are informed that `I` is the source (`{[V]source(I)|V<-Vs}`), and (3) the edges of the graph `(Vs,Es)` are distributed across the ensemble (`{[F]edge(T)|(F,T)<-Es}`). Fact `[X]trans(Ns,P)` encodes the partial distribution of a musical score, where `Ns` is the subsequence being processed by node `X` and `P` is the zero-indexed position of `Ns` from the original score. Rules `fwd`, `ret` and `end` implement the distribution process: `fwd` handles the case where a non-empty sequence of notes (`N:Ns`) reaches the location `X` (`[X]trans(N:Ns,P)`) with outgoing edges (`size(Ys)` > 0). The head `N` of the sequence is recorded with the index position `P` (`[X]note(N,P)`) and the rest of the sequence is propagated through all of `X`'s outgoing edges (`{[Y]trans(Ns,P+1)|Y<-Ys}`). Rule `ret` implements the case when `X` has no successors (`size(Ys)` $= 0$), hence the rest of the sequence `Ns` is wrapped back to the source location `I`. Finally, rule `end` implements the completion of the distribution process: it checks that no non-empty subscore is being processed (`{[X]trans(_:_,_)|X->Xs}` with `size(Xs)=0`) and consumes all empty subscores (`{[_]trans([],_)}`), and then asserts the actuator `completed()`, notifying `I` that the distribution phase has been completed.

Rule `play` is invoked by the trigger `start(S)` at source `I`. It collects all the notes to be played (`{[X]note(N,P)|(X,N,P)->Ms, in(X,Xs)}`) across all the devices in the ensemble (`[I]all(Xs)`) and schedules each to be played `P` seconds from the current time at `I` plus delay of `S` milliseconds (`[X]play(N)@(T+P*1000)` with `T=now()+S`).

Figure 11 shows a snapshot of the state of a four-device



State of the ensemble after executing:

$[\ell_1]dist([A5, B5, G5, G4, D5], Vs, Es)$ followed by $[\ell_1]start(T_0)$
$Vs = \{\ell_1, \ell_2, \ell_3, \ell_4\},\ Es = \{(\ell_1, \ell_2), (\ell_1, \ell_3), (\ell_2, \ell_4), (\ell_3, \ell_4)\}$

Fig. 11. An Instance of Musical Shares

```
 1 rule init  :: [I]initialize(Ps,D)
 2         --o [I]duration(D), [I]livePlayers(Cs,Ms)
 3             {[P]allPlayers(Ps),[P]moderator(I)|P<-Ps},
 4             {[C]notifyCitizen()|C<-Cs}, {[M]notifyMafia(Ms),[M]mafia(Ms)|M<-Ms}
 5             where Ms = pick(Ps,size(Ps)/3), Cs = Ps-Ms.
 6 rule start :: [I]moderator(I) \ [I]start() --o [I]transNight().
 7
 8 rule night :: [I]duration(D), [I]livePlayers(Cs,Ms) \ [I]transNight()
 9         --o {[P]warnNight()@TimeWarn, [P]signalNight()@TimeNight | P<-Cs+Ms},
10             {[M]wakeMafia()@TimeWake | M<-Ms},
11             [I]transDay()@TimeDay, [I]checkVotes()@TimeDay
12             where TimeWarn = now()+1000,     TimeNight = TimeWarn+10000,
13                   TimeWake = TimeNight+5000, TimeDay   = TimeNight+D.
14
15 rule day   :: [I]duration(D), [I]livePlayers(Cs,Ms) \ [I]transDay()
16         --o {[M]warnMafia()@TimeWarn | M<-Ms}, {[P]signalDay()@TimeDay | P<-Cs+Ms},
17             [I]transNight()@TimeNight, [I]checkVotes()@(TimeNight-5000)
18             where TimeWarn = now()+1000, TimeDay = TimeWarn+10000, TimeNight = timeDay+D.
19
20 rule mvote :: [X]mafia(Ms)      \ [X]mafiaVote(C)   --o [I]vote(C).
21 rule cvote :: [X]allPlayers(Ps) \ [X]citizenVote(C) --o [I]vote(C).
22 rule tally :: [I]checkVotes(), {[I]vote(P)|P->Ps}, [I]livePlayers(Cs,Ms)
23         --o [I]livePlayers(Ms-{K},Cs-{K}), {[P]notifyDeath(K)|P<-Cs+Ms}, [I]checkEnd()
24             where K = tally(Ps).
25
26 rule end   :: [I]allPlayers(Ps), [I]livePlayers(Cs,Ms), [I]checkEnd()
27             | size(Ms)>=size(Cs) or size(Ms)=0 --o {[P]notifyEnd(Cs,Ms) | P<-Ps}.
```

Fig. 10. Mafia, an Ensemble-Orchestrated Android Party Game

Android ensemble (ℓ_1, ℓ_2, ℓ_3 and ℓ_4). In this instance, location ℓ_1 has initiated the distribution of the musical score `[A5,B5,G5,G4,D5]` across the ensemble based on the graph `(Vs,Es)` shown in figure. This is followed by executing $[\ell_1]$start(S) which initiates the playing of the score at time $T = T_0 + S$ where T_0 is the current time at ℓ_1.

B. Mafia is a party game played among eight or more players. Initially, a third of the players are secretly assigned to be criminals in the mafia, while the rest are innocent citizens. The game proceeds in day and night cycles: during the night, citizens sleep while criminals roam freely but silently, deciding on one citizen to murder. In the day, all citizens, including the criminals (whose identities are unknown), debate on who the mafia members are and vote on one suspect to execute. The citizens win when all mafia criminals are killed, while the mafia wins when the surviving mafia players outnumber the remaining citizens. In the traditional game, one person in the party acts as the moderator. The moderator does not participate as a player but manages the day and night transitions, while keeping track of the number of surviving players in each team.

In our CoMingle Mafia app, the role of the moderator is no longer assumed by a human, but the ensemble of mobile phones. Day to night transitions proceed as follows: (1) 10 seconds before night time all devices emit a warning beep, prompting players to lay down, hold their devices firmly, face down, and with their eyes closed. (2) Once night has arrived, all devices vibrate at the same time, indicating that players shall fall asleep. (3) 5 seconds later, the mafia devices vibrate once more, indicating that they can begin their evil deeds. Night to day transitions proceed similarly, with a 10 seconds warning buzz to notify mafia members to silently proceed to their original positions and pretend that they have been sleeping through the night.

Figure 10 shows the CoMingle program that orchestrates the Mafia game. We assume that one device, `I`, has been nominated as the moderator.[2] Rule `init` initializes the game from the trigger `[I]initialize(Ps,D)`, where `Ps` is the set of all locations (players), and `D` is the duration of the day and night cycles. Players are partitioned into mafia `Ms` and citizens `Cs` in such a way that `Ms` contains a third of players `Ps` selected arbitrarily (`Ms = pick(Ps,size(Ps)/3)`), while `Cs` is the rest. As the moderator, device `I` is issued facts `[I]duration(D)` and `[I]livePlayers(Cs,Ms)` to keep track of the game state. Each player knows all players `Ps` and that `I` is the moderator (`{[P]allPlayers(Ps),[P]moderator(I)|P<-Ps}`) and is privately notified of its role (`{[C]notifyCitizen()|C<-Cs}` and `{[M]notifyMafia(Ms),[M]mafia(Ms)|M<-Ms}`).

When issued trigger `start()`, the moderator starts off the night transition in rule `start`. Rule `night` is activated by the fact `[I]transNight()`, during which location `I` sets the warning beep event for one second later (`TimeWarn=now()+1000` and `[P]warnNight()@TimeWarn` for each live player `P`). On each device, the actuator `warnNight` calls the device sound library to invoke a beep and the UI to render a countdown sequence from 10 seconds. Rule `night` also schedules the actual night time signal 10 seconds after the warning actuation (`TimeNight = TimeWarn+10000` and `[P]signalNight()@TimeNight` for each live player `P`). The actuator `signalNight` invokes the devices' vibration module, signaling to the user that night has arrived. The next event scheduled by this rule is `{[M]wakeMafia()@TimeWake | M<-Ms}` and this

[2]The owner of this device can still be an active player of the game.

event wakes the mafia members 5 seconds after night time started (TimeWake = TimeNight+5000). Finally the night-to-day transition transDay() and the checking of the votes checkVotes() are scheduled for the end of the night (TimeDay = TimeNight+D). The latter instructs the moderator to count the votes that will have been cast by night's end. The rule day implements the night to day transition and works similarly but with the following difference: the mafia members are given a 10-second warning {[M]warnMafia()@TimeWarn | M<-Ms} to regain their positions and the votes are checked 5 seconds prior to end of the day time.

Rule mvote is where each member X of the mafia proposes a player C to murder. It is activated by the trigger mafiaVote and has the effect of sending its preference to the moderator by means of the fact [I]vote(C). Rule cvote operates similarly, but all players are allowed to cast a vote for whom to execute. The moderator I tallies the vote in rule tally when the fact checkVotes() appears in the rewriting state. It collects all the votes that have been cast in the current period ({[I]vote(P)|P->Ps}) and selects the player K to dismiss using the local function *tally*. It then updates the fact [I]livePlayers(Cs,Ms) by removing K from both Cs and Ms — since these multisets form a partition of the surviving players, only one of them will effectively be updated. Finally, all live players (including K) are informed of this event ({[P]notifyDeath(K)|P<-Cs+Ms}). The rule also adds the fact [I]checkEnd() which checks the game victory condition for the mafia team (*size*(Ms)>=*size*(Cs)) and the citizen (*size*(Ms)=0) and notifies all players of the outcome. It also ends the game by means of the actuator notifyEnd(Cs,Ms).

VI. Related Work

To the best of our knowledge, CoMingle is the first framework to introduce a high-level language construct to orchestrate time-synchronized events across Android mobile devices. However, it draws from work on distributed and parallel programming languages for decentralized systems, which we now review.

CoMingle was 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 and also permits a form of time annotation. However, such time annotations are only used for delaying internal computations, not synchronizing distributed events across the ensemble. Recent refinements [2] extend Meld with comprehension patterns and linearity, but refocused it on distributed programming of multicore architectures.

Microsoft's TouchDevelop [15] is a development environment for the rapid prototyping of Android applications. Like CoMingle, TouchDevelop provides a high-level mobile programming abstraction. However, it mainly focuses on applications that run locally on a device with cloud service integration and touch screen programming interfaces. Device-to-device coordination is not considered.

CoMingle is also inspired by CHR [4], a logic programming language targeting traditional constraint solving problems. Prior to this work, CoMingle extended it with multiset comprehension, explicit locations, triggers and actuators. This papers added time annotations to the list.

VII. Future Work and Conclusion

We have extended the CoMingle language with time annotations. This enable CoMingle to orchestrate time-synchronized events across an ensemble of Android devices. We have described its abstract semantics and implemented a prototype that uses a simplified variant of the TPSN time synchronization protocol. We have tested it on two distributed Android applications that showcase this new feature. In the future, we intend to extend the CoMingle network middleware to accommodate heterogeneous multi-hop networks. This requires more advanced time synchronization protocols (e.g., [3], [13]). We also intend to scale up our experiments to stress-test the CoMingle runtime in maintaining timing requirements of its applications.

VIII. Acknowledgment

This work was made possible by grant JSREP 4-003-2-001 and NPRP 09-667-1-100, from the Qatar National Research Fund (a member of the Qatar Foundation). The statements made herein are solely the responsibility of the authors.

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*, Pasadena, CA, 2009.

[2] 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.

[3] J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. *SIGOPS Oper. Syst. Rev.*, 36(SI):147–163, 2002.

[4] T Frühwirth and F. Raiser. *Constraint Handling Rules: Compilation, Execution and Analysis*. BoD, 2011.

[5] S. Ganeriwal, R. Kumar, and M.B. Srivastava. Timing-sync Protocol for Sensor Networks. In *SenSys'03*, pages 138–149, 2003.

[6] Google. Android Issue Tracker: User App cannot set System Time. https://code.google.com/p/android/issues/detail?id=4581, 2013.

[7] 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*, 2011.

[9] H. Kopetz and W. Ochsenreiter. Clock Synchronization in Distributed Real-time Systems. *IEEE Trans. Comput.*, 36(8):933–940, 1987.

[10] E.S.L. Lam and I. Cervesato. Optimized Compilation of Multiset Rewriting with Comprehensions. In *APLAS'14*, pages 19–38, 2014.

[11] E.S.L. Lam, I. Cervesato, and N. Fatima. Comingle: Distributed logic programming for decentralized mobile ensembles. In *Coordination'2015*, pages 51–66, 2015.

[12] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

[13] M. Maróti, B. Kusy, G. Simon, and Á. Lédeczi. The Flooding Time Synchronization Protocol. In *SenSys'04*, pages 39–49, 2004.

[14] D.L. Mills. Internet Time Synchronization: the Network Time Protocol. *IEEE Transactions on Communications*, 39:1482–1493, 1991.

[15] N. Tillmann, M. Moskal, J. de Halleux, and M. Fahndrich. Touchdevelop — programming cloud-connected mobile devices via touchscreen. Technical Report MSR-TR-2011-49, 2011.